A formal verification tool for Lending Pools

Massimiliano Mirelli

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 24.6.2021

Supervisor

Associate Professor Alberto Lluch Lafuente, Technical University of Denmark
Senior University Lecturer Tommi Junttila, Aalto University

Advisor

Doctoral Candidate James Chiang, Technical University of Denmark

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Massimiliano Mirelli

Title A formal verification tool for Lending Pools

Degree programme Master's Programme in Security and Cloud Computing

Major Security and Cloud Computing **Code of major** SCI3113

Supervisor Associate Professor Alberto Lluch Lafuente, Technical University of Denmark
Senior University Lecturer Tommi Junttila, Aalto University

Advisor Doctoral Candidate James Chiang, Technical University of Denmark

Date 24.6.2021 **Number of pages** 87+47 **Language** English

Abstract

Decentralised Finance (DeFi) applications compose an entire financial ecosystem deployed on the Ethereum blockchain. DeFi applications consist of complex and new protocols whose financial safety is not entirely clear. Besides, their adoption is rapidly growing, hence imperilling an increasingly higher amount of assets. Therefore, an accurate formalisation and verification of their behaviour is essential to deepen the understanding of their safety. A first step in this direction was taken by Bartoletti et al. (arXiv:2012.13230,2020) defining a formal model for the most widespread DeFi protocols: Lending Pools (LP).

The primary aim of this thesis was to develop a verification tool of the LP model. This was achieved by leveraging the Maude verification environment and the MultiVeStA statistical analyser. Maude is a verification toolset enabling to simulate and conduct various analyses on a model specified in the Maude specification language. MultiVeStA is a Java engine enabling statistical analyses via Monte-Carlo discrete-event simulations, such as the ones generated by a model specified in Maude. Thus, a simulator of lending pools was developed in the Maude language and it was integrated with MultiVeStA in order to support several analyses on LP, including reachability analysis, LTL model checking and statistical analyses. The Maude simulator was also validated by a complete suite of test cases.

Furthermore, the proposed tool allows to statistically analyse several parameters of LP, which are fundamental to enhance its safety. In order to illustrate this, a statistical analysis was developed by the means of the MultiVeStA engine. The results of the analysis was that the default parameters, presented by Bartoletti et al., appear to maximise the platform financial safety. Additionally, the verification tool is open to the public under GNU-GPLv2.0 and it is available at https://github.com/MMirelli/maude-lp.

Keywords Formal Methods, Model Checking, Reachability Analysis, Discrete-Event Model Statistical Analysis, Decentralised Finance Safety, Lending Pools Simulator, MultiVeStA Statistical Analyses

Acknowledgements

I am sincerely grateful to my supervisors and advisor. I would first like to thank my thesis advisor James Chiang for being always supportive and ready to clarify any doubt I had during the entire thesis work. Secondly, I would like to show my gratitude to my supervisor Alberto Lluch Lafuente for the very interesting topic he proposed me for this thesis, and his continuous commitment, together with James, in guiding this research project. Additionally, I am very grateful to Tommi Junttila for his precious and detailed feedback aimed at making this work way better than what I could have possibly ever achieved alone.

Besides, my gratitude goes to external researchers I had the great fortune of collaborating with. First, I am thankful to Andrea Vandin, Tenure-track Assistant Professor at Sant'Anna School for Advanced Studies, for having shared his knowledge of MultiVeStA. I would also like to acknowledge Lewis Gudgeon, Doctoral Candidate at Imperial College London, for helping me comprehend the geometric Brownian motion.

Finally, I am profoundly grateful to my spouse, parents{-in-law}, brother, relatives and friends for constantly proving their daily support. This accomplishment would have not been possible without you: thank you!

Espoo, 24.6.2021

Massimiliano Mirelli

With the support of the Erasmus+ Programme of the European Union

Contents

Abbreviations and Acronyms

LP	Lending Pools
DeFi	Decentralised Finance
LFM	Loanable Funds Markets
TTP	Trusted Third Party
DLT	Distributed Ledgers Technologies
GBM	Geometric Brownian Motion
LTL	Linear Temporal Logic
SMC	Statistical Model Checking
GNU	GNU's Not Unix!
GPL	General Public License

1 Introduction

Financial trading has recently shifted to virtual markets, platforms entirely regulated and controlled by novel protocols. Given the considerable amount of funds daily exchanged on such platforms [1, 20], even minor design flaws could determine massive and intolerable losses [32]. Consequently, formal verification of these systems is crucial, in order to ensure their correctness and safe behaviour. This thesis proposes a novel tool capable of verifying financial safety properties for a class of the aforementioned platforms.

This chapter aims at giving a synopsis of the thesis. Firstly, Section 1.1 introduces the research problem. Secondly, Section 1.2 presents the main contributions of this thesis. Lastly, Section 1.3 describes the organisation of the thesis into its sections.

1.1 Context

Ethereum is a recent platform deployed on a novel blockchain [15]. The platform offers a distributed infrastructure to execute applications, i.e. *smart contracts* [25], leveraging the blockchains technology in order to eliminate intermediaries. Smart contracts may implement any type of application, including games [47], e-commerce [5] and financial applications [12, 35]. The ethereum-based financial environment is also called *Decentralised Finance* (DeFi) [54], due to the peer-to-peer nature of the blockchain ethereum relies upon. However, even assuming the security guarantees ensured by the underlying blockchain, DeFi smart contracts have several vulnerabilities latent in their design [45, 58]. Additionally, decentralised finance has recently been employed by a growing community of users. As of June 2021, the growth of the capital locked by DeFi applications has increased by over 370 times since the previous year: from approximately $172mln, on 6 June 2020, to over $64bn, on 6 June 2021 [44]. The increasing value of the assets exchanged by this means motivates the compelling necessity of formally verifying desirable safety properties of such systems. Notwithstanding the increasing interest of several research groups in this area [11, 5, 2, 52, 4, 26], the complexity of such platforms and their recentness constantly arise new interesting research problems. This motivates the development of a new formal verification tool to analyse DeFi systems.

Traditional formal verification techniques, such as model checking, do not directly analyse the implementation of the considered system [18]. Contrarily, they operate on a mathematical representation of the system, the so called *formal model*. The verification tool proposed in this thesis simulates and analyses the *Lending Pools* (LP) model as defined by Bartoletti et al. [7]. LP formally defines the behaviour of the most widespread DeFi applications, namely Aave [12] and Compound [35][1], allowing their users to perform several operations on virtual assets. The two main features of such platforms are lending and borrowing assets, which could suffice to increase the users' profits by various financial practices, including margin trading.

Lending pools formalises highly-distributed systems, where multiple actions could be performed by each user, at any time. In order to capture this complexity, LP

[1] At the time of writing, these are the first and third as per the amount of capital locked [44].

utilises non-determinism. Consequently, the model can be more simply specified in a language capable of expressing non-deterministic events in a simple manner. The Maude project offers a specification language [19] suitable for defining highly concurrent systems. Additionally, Maude provides a very extensive environment for both simulating and verifying the properties of the specified models. Consequently, the lending pools model has been specified in the Maude language. From this moment, the developed Maude specification of lending pools will also be regarded to as the *Maude-based LP simulator*.

However, given the complexity of the modelled systems, the analyses techniques offered by the Maude environment are not sufficient. Specifically, since the system may evolve by following an infinite number of execution paths, the traditional model checking methods result in being either ineffective or unviable. Therefore, the Maude-based LP simulator has been extended to support a more efficient approach, namely statistical analyses. This has been achieved by integrating the simulator with the MultiVeStA statistical analyser [53]. Statistical analyses, despite producing less accurate results, allow to observe the quantitative behaviour of the model, offering statistically-valid results. In the case of lending pools, this approach allows to estimate parameters of the model so to increase its safety. Specifically, an essential safety property of the model is that the number of nonrepayable loans is minimal.

1.2 Contributions

This thesis proposes a Maude-based LP simulator capable of conducting several analyses of lending pools including:

1. reachability analysis;
2. LTL model checking;
3. statistical analysis.

Additionally, the study showcases the usage of the tool by answering a still uninvestigated research question, aiming at an enhancement of the analysed platforms' safety. Precisely, an ad-hoc statistical analysis shows that a choice of the parameters used to instantiate the LP model reduces the amount of nonrepayable loans.

1.3 Structure of the Thesis

The thesis is structured as follows.

Section 2 details the analysed model and gives the necessary prerequisites in terms of both the Maude environment and the developed analyses.

Section 3 describes the developed Maude-based simulator of lending pools, outlining its features and components.

Section 4 showcases a simulation scenario used to conduct the statistical analysis presented at the end of Section 5.

Section 5 illustrates the functionalities of the model and illustrates the results of the statistical analysis whose components were defined in Section 4.

Section 6 elaborates on the contributions of this thesis comparing them with related pieces of research

2 Background notions

2.1 The formal model: Lending Pools

Lending pools, also known as Loanable Funds Markets [55], is a formal model for a class of *Decentralised Finance* (DeFi) protocols [56] allowing their users to lend and borrow assets on a virtual market. Similarly to other DeFi solutions, LP do not require a trusted third party (TTP) in order to operate reliably. In DeFi applications, the guarantees and functionalities offered by a TTP are ensured by *distributed ledgers technologies*, more commonly known as blockchains. Specifically, DeFi applications, including LP, are mainly deployed on the Ethereum blockchain [54].

This section overviews such concepts; specifically, Section 2.1.1 explains distributed ledgers technologies, with a particular focus on blockchains, Section 2.1.2 outlines the basics of Ethereum, and Section 2.1.3 describes lending pools.

2.1.1 Distributed ledger technologies

In the last decade, distributed ledger technologies (DLT) have significantly evolved in a scattered manner, frequently being a source of ambiguity in the literature [57, 3]. This work uses the DLT definition delineated in [46]:

> A DLT [...] is a system of electronic records that enables independent entities to establish a consensus around a shared 'ledger' - without relying on a central coordinator to provide the authoritative version of the records.

In other words, a DLT can be defined as a *fully*-distributed database, whose components are nodes (*peers*) holding replicas of the ledger. Nonetheless, unlike a regular distributed database, a DLT allows the nodes to take part in the so called *consensus* mechanism, entailing the validity of the next possible state. In case all the nodes are given the right to join the decision, the DLT is said *permissionless*; if only a subset of them is allowed, the ledger is known as *permissioned* [14].

Although, several distributed ledgers have been theorized, only one type, blockchains, has been extensively deployed [24]. Since the ledger which Ethereum relies on is a permissioneless blockchain [15], this section only introduces this ledger type. More complete, and still formal, presentations of DLT variants are [14, 24].

A blockchain is often visualized as a linked list of blocks, representing the evolution of the ledger [40]. In the first ever developed blockchain, Bitcoin [39], each block holds the information of several funds transactions between peers. Thus, the current system state is given by all the individual transactions stored on each block of the blockchain. Thus, the integrity the blockchain must maintain is two-fold, in the way that each peer should be able:

1. to prove the validity of a block;
2. to verify the therein transactions.

Bitcoin achieves the former property by storing in each block the hash computed over its content and the hash of its parent block's content. Transaction verification, instead, relies on public key cryptography: each transaction is signed by the private key of the payer, hence it can be verified by any peer in the network, using the payer's public key. Since a blockchain is a DLT, each node of the system stores the whole list of blocks and a consensus algorithm is executed in order to find an agreement on the next valid block, implying a new system configuration.

With respect to this thesis, we define a blockchain as a state transition system, as expressed in [15]. Defining a balance as the amount of a virtual asset held by a blockchain peer, a state of the blockchain transition system is simply the list of its peers' balances. Contrarily, the transition function is defined as a function that, given a state and a set of funds transactions, returns a new valid state (the one with the balances updated accordingly to the transactions). Section 2.1.3 gives a more formal definition of the model this thesis refers to.

2.1.2 Ethereum and smart contracts

Ethereum is often referred to as the direct successor of Bitcoin, since it enables its users to run composable procedures, *smart contracts*, on a secure blockchain-based infrastructure [59]. Smart contracts, first developed by Szabo [50], are programs capable of effectively reproducing the terms of a material contract, having the potential to replace the roles currently played by third parties (namely banks, or similar institutions). Although the blockchain peers are enabled by smart contracts *functions* to interact with them, peers are not allowed to directly control the contracts, which instead execute autonomously [25].

Several applications can be developed leveraging smart contracts, including *Decentralised Finance*. Gudgeon et al. [26] define Decentralised Finance as *a peer-to-peer financial system which leverages distributed ledger-based smart contracts to ensure its integrity and security*. Besides being an effective deterrent to a number of attacks, as observed in [14], Ethereum blockchain ensures that a number of properties are satisfied by DeFi [55].

First, the blockchain consensus mechanism and public key cryptography enables agents to verify any system state, allowing for non-custodial transactions to happen securely. This determines that users are no longer required to trust a third-party custodian, when depositing their funds as collateral.

Second, agents can join the network in a permissionless manner, as they are not bound to any strong identity [15], entailing that a user might have a number of accounts open.

Lastly, as DeFi protocols are implemented by *composing* smart contracts, they can, in turn, be combined in order to extend their functionalities.

Although the previous properties are certainly desirable, they also introduce risks which require to be tackled. In lending pools, those resulting from the first two properties are hindered by a collateralization mechanism, as explained in Section 2.1.3. Contrarily, the composability of smart contracts can cause serious vulnerabilities, still representing an open problem [55].

2.1.3 A class of DeFi applications: lending pools

Lending pools (LP) models a subcategory of DeFi protocols enabling users to borrow and lend virtual assets. At the time of this writing, the smart contracts modelled by LP are the most utilised class of DeFi protocols, with the majority of them being deployed on Ethereum [44]. Their functioning differs from both regular lending and peer-to-peer one [27]. In fact, these protocols are dissimilar from the former, due to their decentralised nature, as described in Section 2.1.1. Similarly, they can neither be considered fully peer-to-peer lending applications, as assets are not directly borrowed and lent by agents. Instead, deposited funds are pooled and lent on-demand to borrowers, only if they possess enough collateral (i.e. only if their account is overcollaterized). As Ethereum does not provide strong identities, but pseudonyms [15], users' actions are difficult to be regulated under a jurisdiction, which makes collateralization the main protection mechanism against adversarial behaviours [41]. According to this mechanism, an agent can only borrow a quantity of tokens worth less than the amount of collateral they deposited. This mechanism and others, including interest rates, is in place in order to incentivize borrowers to repay their loans.

The remaining part of this section details the lending pools model formalised by Bartoletti et al. [7]. They give an accurate operational description of the semantics of LP protocol actions, based on the analysis of two major lending pools protocols: Compound [35] and Aave [12].

The basic components of their model are *agents* and *cryptoassets*. LP agents are the rational entities taking part in the protocol. Contrarily, LP cryptoassets are token types, each representing a different virtual currency. Bartoletti et al. distinguish two classes of token types: *free tokens* and *minted tokens*, denoted respectively by the sets $\mathcal{T}_f = \{\tau_i\}_{i\in[1\ldots k]}$ and $\mathcal{T}_m = \{\tau'_i\}_{i\in[1\ldots k]}$, where k is the number of cryptocurrencies available on the pool. The set of all tokens $\mathcal{T}$ is defined as $\mathcal{T} := \mathcal{T}_f \cup \mathcal{T}_m$. The substantial difference between these classes of token types is that free tokens have a value established by external markets, whereas minted tokens are assets coined by the protocol, hence holding value only in a specific LP environment. In other words, minted tokens can be considered as loyalty credits held by the agents actively joining the protocol. In fact, minted tokens are granted by the protocol to the agents in return for free tokens, hence each minted token τ' corresponds to a free token τ, also called its underlying token[2].

Given agents and assets, the LP model can be represented as a transition system where each state, or configuration Γ, is of the form $\Gamma := \sigma \mid \pi \mid p$. First, the *wallets* function σ is defined as the function (2.1) storing each agent's balance of free tokens. For instance, the wallet of a generic agent A is expressed by the partial function σ_A, and the balance of its τ-typed tokens by $\sigma_A(\tau)$.

$$\sigma : \mathcal{A} \to (\mathcal{T} \to \mathbb{R}_0^+) \tag{2.1}$$

The second configuration component, the *pool* π, is a triple storing the current lending state of the LP. It is composed by three partial functions as in (2.2): π_f

[2]The underlying token of τ' is also denoted as $u_\pi(\tau') = \tau$.

storing the amount of free tokens deposited on the pool, π_l memorising the loans each agent owes to the pool and π_m keeping track of the minted tokens (also called the *collateral* or credits) purchased from the pool.

$$\begin{aligned} \pi := (\quad & \pi_f, \quad \pi_l, \quad \pi_m \quad), \quad \text{where} \\ & \pi_f : \quad \mathcal{T}_f \to \mathbb{R}_0^+ \\ & \pi_l : \quad \mathcal{A} \to (\mathcal{T}_f \to \mathbb{R}_0^+) \\ & \pi_m : \quad \mathcal{T}_f \to (\mathcal{T}_m \times \mathbb{R}_0^+) \end{aligned} \tag{2.2}$$

The third configuration component is the price partial function p, defined in (2.3). The price function stores the price of each free token, available in the pool.

$$p : dom(\pi_f) \to \mathbb{R}_0^+ \tag{2.3}$$

In order to make the notation simpler and more lightweight, some additional notation should be introduced. Given a partial map f, the notation $f\{v/x\}$ indicates a point-wise update of f at point x, as specified in (2.4).

$$f\{v/x\} := \begin{cases} f\{v/x\}(x) = v \\ f\{v/x\}(y) = f(y) \text{ if } y \neq x \end{cases} \tag{2.4}$$

Additionally, the partial binary map $\circ$ is introduced so to add elements to the domains of the partial functions previously defined. Hence, given a partial map $f : \mathcal{T} \to \mathbb{R}_0^+$, a token type $\tau \in \mathcal{T}$ and a partial binary operation $\circ : \mathbb{R}_0^+ \times \mathbb{R}_0^+ \to \mathbb{R}_0^+$, the partial map $f \circ v : \tau$ is defined in (2.5).

$$f \circ v : \tau := \begin{cases} f\{f(\tau) \circ v/\tau\} \text{ if } \tau \in \text{dom}(f) \text{ and } (f(\tau) \circ v) \in \mathbb{R}_0^+ \\ f\{v/\tau\} \text{ if } \tau \notin \text{dom}(f) \end{cases} \tag{2.5}$$

In order to describe the model evolution, some additional definitions shall be given. The following LP components may rely on the whole configuration, Γ, or some of its components. This dependency is indicated by the means of subscripts. For instance, defining the function F as F_X implies that the value returned by the function F is based on the state of the X component of Γ.

Firstly, the concept of loan and minted value held by a specific user is explained. Intuitively, the value functions return the current value of respectively the minted tokens owned and the free tokens borrowed by an agent. More precisely, the loan value V_Γ^l and the minted value V_Γ^m functions for a generic agent A are defined in Equations (2.6) and (2.7)[3].

$$V_\Gamma^l(A) := \sum_{\tau \in \mathcal{T}_f} (\pi_l(A))(\tau) \cdot p(\tau) \tag{2.6}$$

[3] $ER_\pi(\tau, \tau')$ is the exchange rate of the minted token τ' into τ. For brevity, this is not defined, the interested reader can refer to [7], Section 3.4.

$$V_\Gamma^m(A) := \sum_{\tau' \in \mathcal{T}_m} \sigma_A(\tau') \cdot ER_\pi(\tau', \tau) \cdot p(\tau) \tag{2.7}$$

Secondly, the collateralization function is an essential indicator of agents' lending safety. In fact, a collateralization below a given threshold ($C_{\min}$) entails an agent to be liquidated and hence to incur in a financial loss, as detailed later. The collateralization of an agent A is defined as (2.8).

$$C_\Gamma(A) = \frac{V_\Gamma^m(A)}{V_\Gamma^l(A)} \tag{2.8}$$

Having introduced the necessary model components, the model dynamics can now be outlined. In fact, the services offered by the smart contracts (so called functions) are also formalized in [7]. These functionalities of smart contracts are effectively represented by agents' actions (or *rules*) having a precise formal semantics. Accordingly to the notation in [7], a rule named "r" and executed by agent A with n input parameters z^n is indicated by $\mathrm{r}_A(z^n)$[4]. Given agents A and B, LP rules are summarised in Table 2.1.

In lending pools, a committed rule entails the system to change configuration as per its semantics. A system transition from state Γ to Γ' via the $\mathrm{r}_A(z^n)$ action is expressed by the notation: $\Gamma \xrightarrow{\mathrm{r}_A(z^n)} \Gamma'$. Consequently, a specific instantiation of the LP model could be represented by a transition system (S, I, R, AP, L) as in Table 2.2.

In order to specify more precisely the semantics of an LP action, the liquidate action from [7] is shown in Figure 2.1 and its semantics are explained. According to the standard operational semantics syntax [43], the transaction below the solid line is executed if the eleven preconditions are satisfied. The essential preconditions to understand the rule are ④, ⑧, ⑨, ⑩ and ⑪.

① $\tau' \in \mathcal{T}_m$	② $\sigma_A(\hat{\tau}) \geq v$	③ $\pi_l(B)(\hat{\tau}) \geq v$
④ $v' := v \cdot \frac{p(\hat{\tau})}{p(u_\pi(\tau'))} \cdot r_{\text{liq}}$	⑤ $\sigma_B(\tau') \geq v'$	⑥ $\pi'_f := \pi_f + v : \hat{\tau}$
⑦ $\pi'_l := \pi_l\, B - v : \hat{\tau}$	⑧ $\sigma'_A := \sigma_A - v : \hat{\tau} + v' : \tau'$	⑨ $\sigma'_B := \sigma_B - v' : \tau'$
⑩ $C_{\sigma\|\pi\|p}(B) < C_{\min}$	⑪ $C_{\sigma'\|\pi'\|p}(B) \leq C_{\min}$	

$$\sigma \mid \pi \mid p \xrightarrow{\mathsf{Liq}_A(B, v:\hat{\tau}, \tau')} \sigma\{\sigma'_A/A\}\{\sigma'_B/B\} \mid (\pi'_f, \pi'_l, \pi_m) \mid p$$

Figure 2.1: The liquidate rule in [7].

④ - computes the reward for the liquidating agent. This is based on the liquidated amount v and the reward factor r_{liq}. Here the idea is that A, by repaying part of B's loan, is reducing the likelihood of the protocol to become illiquid. As a result, this behaviour is incentivized by the platform by setting the aforementioned

[4]This can be also defined as an action of a given arity n.

$\mathrm{Trf}_A(B, v : \tau)$	A transfers v free-tokens of type τ from its wallet to B's one.
$\mathrm{Mtrf}_A(B, v : \tau')$	A transfers v units of minted token τ' to B, as long as A remains overcollateralized.
$\mathrm{Dep}_A(v : \tau)$	A deposits v free-tokens of type τ from its wallet to the pool. Subsequently, the pool coins v' units of τ', with v' computed so to incentivize deposits only if the LP is lacking free tokens.
$\mathrm{Rdm}_A(v : \tau')$	A redeems v units of the minted token τ', as long as A's collateralization is greater than a threshold ($C_{\min}$) and LP holds enough tokens of type τ'.
$\mathrm{Bor}_A(v : \tau)$	A borrows v units of a free token τ, assuming it has enough collateral.
$\mathrm{Rep}_A(v : \tau)$	A repays v units of its loan in the free token τ to the LP.
$\mathrm{Liq}_A(B, v : \hat{\tau}, \tau')$	A (liquidator) liquidates a variable amount of B's (borrower's) minted tokens τ', by paying v units of free tokens $\hat{\tau}$. Notably, $\hat{\tau} \in \mathcal{T}_f$ is in general different than τ, the underlying token of $\tau' \in \mathcal{T}_m$. This action can be executed only if the B's collateralization is below $C_{\min}$, meaning B is undercollaterized.
Int	The LP underlying smart contract sets a new interest rate for specific token types. This is a means for the system to disincentivize borrowers from postponing their loans repayment.

Table 2.1: Summary of lending pools actions from [7]

S	a countably infinite set of states (or configurations) of the form $\Gamma := (\sigma \mid \pi \mid p)$, as previously defined.
I	a set of initial states, each $\Gamma^i := (\sigma^i \mid \pi^i \mid p^i)$ comprising the initial agents' wallets function σ^i, an empty pool $\pi^i := \{\pi^i_f,\ \pi^i_l,\ \pi^i_m\}$ s.t. $dom(\pi^i_f) = dom(\pi^i_l) = dom(\pi^i_m) = \emptyset$, and the initial assets price function p^i.
$R \subseteq S \times S$	a transition relation s.t. for each state $\Gamma, \Gamma' \in S,\ (\Gamma, \Gamma') \in R$ iff there is a rule $\mathrm{r}_A(z^n),$ executable in Γ and $\Gamma \xrightarrow{\mathrm{r}_A(z^n)} \Gamma'$.
AP	a set of atomic propositions.
L	a labelling function associating each configuration $\Gamma \in S$ to a set of propositions P . $P \in 2^{AP}$.

Table 2.2: Transition system components

reward to a value strictly higher than 1. According to Bartoletti et al. the default r_{liq} equals 1.1.

(8) and (9) - update the involved agents' wallets, A repays v units of B's loan in $\hat{\tau}$ and is compensated with v' units of τ'

(10) - ensures that the rule is executable only if B's collateralization is less than $C_{\min}$, which is assigned 1.5 as default value by Bartoletti et al.. This rule is the reason why agents' collateralization should be at least $C_{\min}$, so to avert the risk of being liquidated and incurring in the loss of the liquidation reward r_{liq}.

(11) - prevents A from seizing a higher collateral amount than the one required for B to be considered safe (i.e. $C_\Gamma(B) \geq C_{\min}$). It should be noted that Aave [13] and Compound [42] limit the amount of repayable loan by a percentage factor *Maxliq*.

Figure 2.2 illustrates the transition system for a simple running example, where three liquidate actions are executed. The figure shows six possible traces all originating from Γ_0^i and having $\Gamma_{3,1}$ as final state. Each configuration in the figure is defined by a row in Table 2.3. Additionally, transitions, namely Liq actions performed by D, are indicated by different colours depending on the liquidated borrower in both the transition system and the table. Notably, assuming $C_{\min} = 1.5$ and $r_{\text{liq}} = 1.1$, all borrowers in Γ_0^i, A, B and C, are undercollaterized. Specifically, A is marginally undercollaterized since $C_{\Gamma_0^i}(A) = 1.25 > 1.1 = r_{\text{liq}}$, while B and C are strongly undercollaterized, being both $C_{\Gamma_0^i}(B)$ and $C_{\Gamma_0^i}(C)$ below 1.1. This allows D to seize the entire B and $C's$ collateral, as evident from $\Gamma_{3,1}$ in Table 2.3. Contrarily A's collateralization is restored to $C_{\min}$.

As an example, the transition $\Gamma_0^i \xrightarrow{\text{Liq}_D(B,91:\tau_1,\tau_0')} \Gamma_{1,2}$ is explained. In this case, agent D repays 91 units of τ_1, seizing $91 \cdot r_{\text{liq}} \approx 100$ units of τ_0' from agent B. Notably, this also affects π, in a way that the funds in τ_1 are incremented by 91 units, as illustrated by $\pi_f(\tau_1)$, while B's loan is decremented by 91 units, as shown by $\pi_l(B)(\tau_1)$. Contrarily, π_m is not modified by the transaction, as the 100 units of minted tokens τ_0' are simply transferred from B's wallet to D's one.

2.2 The Maude specification language

This section introduces the Maude language [19], some Maude utilities and conventions. The Maude language is a declarative algebraic specification language, commonly employed for either specifying or programming highly concurrent software systems. Additionally Maude encompasses a wide range of formal verification tools [23, 28], including a command line interpreter tool [19], which is the one employed in this study.

Maude specifications are composed by two types of basic blocks: *functional* and *system* modules. Thus, Sections 2.2.1 and 2.2.2 define functional modules and system modules, respectively. In parallel with the description of functional and system modules, also the relevant Maude interpreter commands are explained. Subsequently,

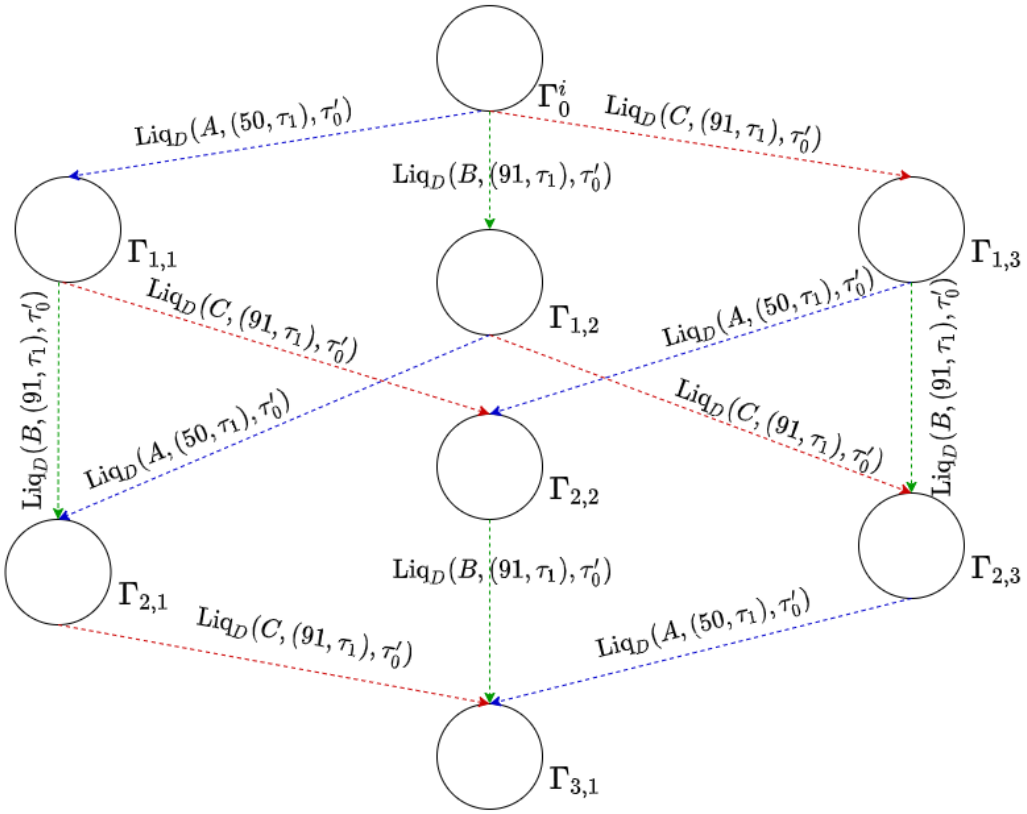


Figure 2.2: Example of transition system produced by the initial configuration (Γ_0^i) holding three liquidate actions, executed by agent D. The configurations of the graph are defined in Table 2.3.

Γ	π_f	π_l			σ_A		σ_B		σ_C		σ_D			C_Γ		
		A	B	C												
	τ_1	τ_1	τ_1	τ_1	τ_1	τ_0'	τ_1	τ_0'	τ_1	τ_0'	τ_1	τ_0'	τ_1'	A	B	C
Γ_0^i	195	80	100	125	80	100	100	100	125	100	500	0	500	1.25	1	0.8
$\Gamma_{1,1}$	**245**	**30**	100	125	80	**45**	100	100	125	100	**450**	**55**	500	**1.5**	1	0.8
$\Gamma_{1,2}$	**286**	80	**9**	125	80	100	100	**0**	125	100	**410**	**100**	500	1.25	**0**	0.8
$\Gamma_{1,3}$	**286**	80	100	**34**	80	100	100	100	125	**0**	**410**	**100**	500	1.25	1	**0**
$\Gamma_{2,1}$	**336**	**30**	**9**	125	80	**45**	100	**0**	125	100	**359**	**155**	500	**1.5**	**0**	0.8
$\Gamma_{2,2}$	**336**	**30**	100	**34**	80	**45**	100	100	125	**0**	**359**	**155**	500	**1.5**	1	**0**
$\Gamma_{2,3}$	**377**	80	**9**	**34**	80	100	100	**0**	125	**0**	**318**	**200**	500	1.25	**0**	**0**
$\Gamma_{3,1}$	**427**	**30**	**9**	**34**	80	**45**	100	**0**	125	**0**	**268**	**255**	500	**1.5**	**0**	**0**

Table 2.3: Configurations of the transition system in Figure 2.2. For simplicity, the price function p is assumed to be constant such that $p(\tau_0) = p(\tau_1) = 1$ in every configuration Γ. In this case $C_{\min} = 1.5$, $r_{\text{liq}} = 1.1$ and $Maxliq = 1.0$.

Section 2.2.3 introduces the notation used in order to graphically represent Maude modules importation hierarchies. Finally, Section 2.2.4 introduces parameterised modules by illustrating parameterised sets in Maude.

2.2.1 Functional modules

A Maude functional module can be informally viewed as a set of types and their properties. The module `MAP{X::TRIV,Y::TRIV}` (in Listing 2.1) is an example of a functional module parameterised on its key and value types `X` and `Y`, respectively, and defining a map (or dictionary) in Maude. In the rest of this section, the module is used to introduce the basic syntax and semantics of the constructs composing a functional module.

```
1   fmod MAP{X::TRIV, Y::TRIV} is
        ⋮              ⋮              ⋮

3   --- Sorts and subsorts
4   sorts Entry{X,Y} Map{X,Y} .
5   subsort Entry{X,Y} < Map{X,Y} .
6
7   --- Variables
8   var D : X$Elt .
9   vars R R' : Y$Elt .
10  var M : Map{X,Y} .
11
12  --- Operators
13  op emptyM : -> Map{X,Y} [ctor] .
14  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor prec 50] .
15  op _;_ : [Map{X,Y}] Entry{X,Y} -> [Map{X,Y}]
16  [ctor assoc comm id: emptyM prec 51 format (d r os d)] .
17  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
18
19  --- Equations and memberships
20  eq insert(D, R, M) = (M ; D |-> R) .
21  mb M ; (D |-> R) : Map{X,Y} .
        ⋮              ⋮              ⋮

23  endfm
```

Listing 2.1: Definition of a map (or dictionary) in Maude, extract from Listing A.2, Line 174, originally defined in `prelude.maude`, as better explained in appendix.

As it is customary in algebraic specification languages, abstract data types are called *sorts*. Each sort `S` is identified by a capitalized name and it is declared as in the example in Line 4. Line 4 is an example of multi-sorts declaration, defining both a sort for a single key-value pair, `Entry{X,Y}`, and one for a set of such pairs, `Map{X,Y}`. Subsequently, Line 5 shows how to define a basic sort hierarchy, with `Entry{X,Y}` being a subsort of `Map{X,Y}`. Subsorts are a very powerful feature of Maude, allowing to define subtypes sharing the properties held by their supertypes

and extending them by defining additional ones. For instance, in this case it is natural to define the key-value pair type as a subtype of the type identifying a key-value pair set.

In order to specify properties of a given sort, the terms used in those properties ought to be declared. In Maude, terms are the atomic blocks of the specified language syntax. Terms may be declared by two statements, either a *variable* or an *operator* statement. A variable statement instantiates a new term valid only in the current module, whereas an operator statement is available also in super-modules[5]. Variable declaration is illustrated in Lines 8 to 10, defining the variables `D`, `R` and `R'` as terms of two generic[6] sorts depending on `X`[7] and `Y`[8], while `M` as a term of sort `Map{X,Y}`. From this point onwards, the fact that a term `t` belongs to a sort `S` is indicated by the notation `t:S`. Variables of a sort `S` can be thought as terms representing any piece of syntax (even specific operators) of sort `S`. Their usage will be clearer, after having introduced equation and membership statements.

An operator is a statement defining the syntax of a valid Maude term. Lines 13 to 17 present four operator instances. The first one (Line 13) is also called a constant, as it does not support input terms, whereas the remaining (Lines 14, 15 and 17) are non-constant operators. Given the functional module in Listing 2.1, a map from strings to integers can be instantiated by importing the module `MAP{String,Int}`[9]. In modules importing `MAP{String,Int}`, both the constant operator `emptyM` and the term `"key0" |-> 0` are interpreted as valid terms of sort `Map{String,Int}`. Notably, the sort of the first input parameter of the operator `_;_` (Line 15) as well as the output parameter are surrounded by square brackets. When applied to sorts, this syntax indicates *kinds*, i.e. particular types used to specify that a term is not well-formed. Precisely, a term `t` necessarily has a kind `[S]`, while it is not of sort `S` unless it is well-formed. As a result, the operator `_;_` (Line 15) generates terms which are not necessarily well-formed. Additionally, operators are equipped with attributes, a list of Maude keywords contained in square brackets specifying the operators properties, as shown in Line 16. For instance, the attribute `ctor`, assigned to operators in Lines 13 to 15, indicates that those operators are constructors of a term. Constructors define the syntax of basic terms in Maude, i.e. their canonical form. The canonical form of a given term is fully comprehended only after having introduced the next fundamental construct: equations.

Properties of sorts are expressed via *equations* and *memberships*. Equations allow to resolve the left-hand side term into its right-hand side equivalent term. For instance, using the module `MAP{String,Int}`, the map consisting of the key `"key0"` associated to the value `0` can be obtained with the `insert` operator Line 17, whose semantics is specified by the equation in Line 20. The simple map is expressed by the term `insert("key0", 0, emptyM)`, whose canonical form is `"key0" |-> 0`. It

[5] Section 2.2.3 discusses modules composability and importing modes.
[6] Maude parameterised programming is more thoroughly explained in Section 2.2.4.
[7] I.e. `X$Elt`, sort of the view `TRIV` as defined in Listing 2.10 and explained in Section 2.2.4.
[8] I.e. `Y$Elt`.
[9] `String` and `Int` are sorts defined in the native Maude library, `prelude.maude`.

is now clear that the canonical form of a term is an equivalent[10] term specified by the means of constructors. Remarkably, in this case, the term `emptyM` does not appear in the canonical form, as the operator `_;_` is defined with the attribute `id: emptyM`. That attribute defines two different maps to be equal modulo the terms `emptyM` they possibly contain. Consequently `emptyM ; "key0" |-> 0` is equivalent to `"key0" |-> 0`.

Finally, memberships define a term to be of a specific sort. For instance, by declaring Line 21, the term `"key0" |-> 0`, initially of kind `[Map{String,Int}]`, is casted to `Map{String,Int}`.

Membership and equational statements may be conditional, meaning that their validity depends on a boolean condition `c`. In other words, if the term `t` does not satisfy `c`, then the conditional membership or equation having condition `c` do not modify the term `t`. The syntax of conditional equations and memberships is omitted, as it is very similar to the one of conditional rules, specified in Listing 2.3.

Maude (conditional) equations and memberships are the statements determining the results generated by the `reduce` (or `red`) Maude command. The `reduce` command syntax and semantics are briefly[11] explained below.

```
- reduce in mod : t .
```

`mod` is a valid module name;

`t` is the term to be reduced to its corresponding canonical form as specified by constructor operators.

The command simplifies the input term `t`, by applying the equational and membership simplification corresponding to the equation and membership statements in `mod`.

Formal definition More formally, a Maude functional module represents a theory in membership equational logic, i.e. a Horn [12]logic having its atomic clauses in the form of (possibly conditional) equations or memberships as expressed in (2.9) [19].

$$a_0 \quad \text{if} \quad \bigwedge_{i=1}^{n} a_i, \qquad \text{for } \{a_i\}_{i\in[0..n]} \text{ atomic clauses} \tag{2.9}$$

Clavel et al. [19] define a membership equational logic theory as $\mathcal{E} := (\Sigma,\ E \cup A)$ where:

- Σ is a set of sorts and kinds;

[10] W.r.t. the module equations.

[11] A more extensive explanation can be found in [19], Section 23.

[12] A clause in Horn logic is defined as a disjunction of literals, having at most one positive literal, i.e. in propositional logic $u \vee \bigvee_{i=1}^{n} \neg q_i$ for $u, \{q_i\}_{i\in[1,n]}$ literals [37]. Note that this writing is equivalent to $u \leftarrow \bigwedge_{i=1}^{n} q_i$, recalling the form of (2.9).

- E contains the theory equations and memberships;

- A are the axiomatic properties referred to elements of E.

Thus, the aforementioned equational simplifications can be formalized by *term reductions* denoted as $t \rightarrow_E^* t'$, meaning that t is reduced to a an irreducible term t' (i.e. *canonical form*), by applying equations and memberships in E for a number of times greater or equal than zero.

The peculiarity of functional modules, which is mostly influencing their use and application is that E must be *terminating* and *confluent*. E is defined to be terminating when an infinite sequence of equational simplifications as in (2.10) does not exist.

$$t \rightarrow_E t_1 \rightarrow_E t_2 \cdots \tag{2.10}$$

E is said to be confluent whenever (2.11) holds.

$$\text{if } \exists\, p_1, p_2 \text{ s.t.} \begin{array}{l} p_1 := t \rightarrow_E^* t_1 \\ p_2 := t \rightarrow_E^* t_2 \end{array} \quad \text{then } \exists\, p_1', p_2' \text{ s.t.} \begin{array}{l} p_1' := t_1 \rightarrow_E^* t' \\ p_2' := t_2 \rightarrow_E^* t' \end{array} \tag{2.11}$$

Termination and confluence ensure the reduction of any term to be unique, whatever the application order of its simplifications is.

2.2.2 System modules

In Maude, non-deterministic systems, such as the one in Section 2.1.3, are specified by means of system modules by defining a syntactical Maude construct called rewrite rules. Rewrite rules implement the behaviour of non-deterministic transitions between two states of the specified system. The states of the system are implemented by the `Configuration` sort, defined in the `CONFIGURATION` system module. Consequently, this section introduces system modules by describing the `Configuration` sort and the typical construct of system modules: rewrite rules.

A `Configuration` is regarded to as a multiset[13] of `Objects` and `Messages`[14], as illustrated by the `__` operator attributes in Listing 2.2.

```
1  mod CONFIGURATION is
       ⋮              ⋮              ⋮

3  sorts Oid Cid Object Msg Portal Configuration .
4  subsort Object Msg Portal < Configuration .
5  op <_:_|_> : Oid Cid AttributeSet -> Object
6             [ctor object] .
7  op none : -> Configuration [ctor] .
```

[13] A collection which might exhibit repeats (similarly to a list), although the relative order of its components is unimportant (like a set).

[14] `Msg` in the specifications, here `Message` is used for clarity.

```
8   op errorConfig : -> [Configuration] .
9   op __ : [Configuration] [Configuration] -> [Configuration]
10        [prec 53 ctor config assoc comm id: none] .
        ⋮              ⋮              ⋮

12  endm
```

Listing 2.2: Maude definition of the generic `Configuration` used in the LP specifications. Extract from Listing A.7, Line 1, originally defined in `prelude.maude`, as better explained in appendix.

An `Object` (Line 5) is used to identify an instance of a specific type in Maude, hence it consists of three components: the object identifier (of sort `Oid`), the class identifier (sort `Cid`) and an attribute set (sort `AttributeSet`), i.e. the data stored in the object. Thus, `Object`s are declared with the following syntax `< kO : kC | kA >`, where `kO`, `kC` and `kA` are constant operators returning terms of sorts `Oid`, `Cid` and `AttributeSet`, respectively.

`Message`s (Line 3) are used as a communication means by `Object`s, so that they can interact with each other. A `Message` is the component of a `Configuration` making it dynamic, as its consumption entails a change in the current `Configuration`.

Rewriting rules are the statements characterising system modules and distinguishing them from functional ones. Rewriting rules are declared according to the syntax in Listing 2.3, where `lh` is the left-hand side term, `rh` is the right-hand side term and `c` is the rule condition. Here the semantics is slightly different than the one used for equations and memberships: if the term matches `lh` and `c` is satisfied, then the rule *may* modify the term.

```
crl lh => rh if c .
```

Listing 2.3: Syntax for a Maude conditional rewrite rule.

Maude (conditional) rewriting rules are the statements determining the results generated by the `rewrite` Maude command. The `rewrite` command syntax and semantics are briefly[15] explained below.

```
- rewrite {[n]} {in mod :} t .
```

`mod` is a valid module name;

`t` is the term to be rewritten;

`n` number of rewriting rules to be applied.

The command rewrites the input term `t`, by applying `n` rewriting steps. Each rewriting step will modify `t` according to a randomly selected (conditional) rewrite rule whose left-hand side matches[16] `t`. It should be noted, that `rewrite` applies

[15] A more extensive explanation can be found in [19], Section 23.

[16] In other words, it is equivalent, given the equations in module `mod` and its submodules.

the rewrite rules on the terms in their canonical form. Consequently, equation and membership statements modify the input term before rewrite rules are performed. Lastly, arguments `[n]` and `mod` are optional, as indicated by the curly brackets. In case `[n]` is not used `t` is rewritten for as many steps as possible, while if `mod` is not indicated, the rewrite is assumed to be performed in the last declared module.

The semantics of a rewrite rule is better understood by the rule in Listing 2.4 applied via a `rewrite` command to the simple configuration in Listing 2.5. Listing 2.4 illustrates the way a Maude `Message`, in this case `transfer(A0, B0, (v, tau))`, is consumed and entails a change of the system state. The rule states that an agent, whose id is `A0`, is transferring `v` units of `tau` to another agent, whose id is `B0`. Consequently, `v` units of `tau` are subtracted from `A0`'s wallet, represented by `sigmaA`[17], and added to `B0`'s one, `sigmaB`, as shown in the right-hand side of the rule. Listing 2.5 shows an example of the rule application to a simple configuration, via a `rewrite` command. The `rewrite` causes agent `A` to transfer `5.0` units of `tau(0)` to agent `B`.

```
op transfer : Agent-Id Agent-Id
              Pair{Float0+, Token} -> Msg [ctor] .

crl [transfer] :
       < A0 : agState | * sigmaA >
       < B0 : agState' | * sigmaB >
       transfer(A0, B0, (v, tau))
       => < A0 : agState | * sigmaA - v : tau >
          < B0 : agState' | * sigmaB + v : tau >
    if sigmaA[tau] >= v .
```

Listing 2.4: Simplified Maude definition of the Trf rule, as defined in Section 2.1.3. Adapted from Listing A.7, Line 1230

```
rewrite < A : agState | * tau(0) |-> 10.0 >
        < B : agState' | * tau(0) |-> 0.0 >
        transfer(A0, B0, (5.0, tau(0))) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Configuration :
        < A : agState | * tau(0) |-> 5.0 >
        < B : agState' | * tau(0) |-> 5.0 >
```

Listing 2.5: Example of a `rewrite` command rewriting a `transfer Message`. The module containing Listing 2.4 is assumed to be the last-loaded module.

At this stage, it is essential to realise that a configuration may be viewed as a sentence in a given formal language. In the case of Maude, the language is expressed by equational and membership statements as explained in Section 2.2.1.

[17]The operator `*_` is the constructor of an `AttributeSet`, as specified in Listing A.7, Line 167.

Formal definition More formally, a Maude system module corresponds to a theory in rewriting logic defined as $\mathcal{R} := (\mathcal{E}, \phi, R)$ [19] where:

- $\mathcal{E}$ is the underlying membership equational logic $(\Sigma,\ E\ \cup\ A)$, as defined in Section 2.2.1;
- ϕ is the set of kind properties (equations or memberships) whose arguments cannot be modified by rewriting rules;
- R is a set of rewriting rules.

Consequently, Maude system modules are a generalization of functional modules. As noticeable from the definition of $\mathcal{R}$, system modules not merely define types of terms and their properties, but they also describe how terms are to be modified by means of rewriting rules. Rewriting rules are Maude constructs very similar to previously introduced equations, in the way that they are pattern matching rules allowing a specific kind of reductions, so called rewrites, to take place. However, a rewrite differs from an equational simplification reduction in that rewriting rules are neither necessarily terminating nor necessarily confluent. Thus, the application of multiple rewrite rules is not guaranteed to simplify the initial term to its canonical form. Therefore, rewrite rules are the fundamental construct for modelling dynamic and (possibly) nondeterministic behaviours of a system.

2.2.3 Composing modules

In Maude, modules are composable, formally yielding to sub and super theories. Three different modes of importing a module exist: `protecting`, `extending` and `including`. As these mainly affect the interpretation of the Maude theorem proving tools[18], which are not employed in this study, the modes are not explained here.[19] Nonetheless, the graphical notation used to indicate the different importation modes shall be presented, in order to better understand the discussion on the developed specification in Section 3. The notation used is inspired by [19], Section 6.3.1.. Each box represents a module, with a blue-coloured frame denoting a functional module and a red-coloured a system module. The relative position of a box with respect to the connected ones signifies whether it is a super-module (importing the connected) or a sub-module (imported by the connected). Specifically, if module `M1` is placed above a module `M2` and `M1`, `M2` are connected, `M1` imports `M2`. Additionally, edges connecting boxes represent the modules importation type: dashed edges indicate `including`, single-solid edges indicate `extending` and double-solid edges indicate `protecting`. The developed specifications uniquely utilise `including` and `extending` modes, as they make the most basic assumptions of the imported model. Precisely, `extending` is employed in the modules defining the testing terms, as those could cause the importing module to have additional ground terms w.r.t. the imported, hence possibly invalidating the `protecting` assumption explained in [19], Section 6.1.1.

[18] Such as the Inductive Theorem Prover (ITP) [28].
[19] The interested reader can find a thorough discussion on this in [19], Section 6.1.

2.2.4 Parameterised modules

Module parameterisation is a Maude feature allowing to declare new datatypes by reusing their operators logic, hence increasing reusability. Parametric modules are either functional or system modules[20], declared by the syntax in Listing 2.6.

```
fmod M {L0::T0, L1::T1, ..., Ln::Tn} is
--- sorts, operators, equations or memberships
endfm
```

Listing 2.6: Parameterised module syntax.

In the listing, `M` indicates the module name, frequently denoted by an uppercase string. Contrarily, `L0::T0, L1::T1, ..., Ln::Tn` define the module's input parameters, or interface. A single parameter `Li::Ti` comprises of a label `Li` and a theory `Ti`. The parameter label `Li` is used to identify the parameter inside the module `M`, whereas `Ti` indicates the parameter *functional theory*. Functional theories follow the syntax in Listing 2.7 and may contain any Maude statement defined in Section 2.2.1. A Maude theory `Ti` can be thought as a set of constraints which must be satisfied by a module bound to `Ti`. In fact, a parameterised module `M` is instantiated by binding a functional module `FMi` to each parameter `Li::Ti` such that `FMi` satisfies the properties defined by `Ti`. This association is performed by a *view*, a Maude construct, used to bind functional modules to the respective theories. Views consist of simple statements mapping functional modules' sorts and operators to compatible theories' sorts and operators.

```
fth FT is
--- sorts, operators, equations or memberships
endfth
```

Listing 2.7: Functional theory syntax.

In order to clarify the instantiation of a parameterised module, the rest of the section discusses how to define a set of integers in Maude. This is achieved by only employing Maude native modules, shipped with Maude v3.0 in the library `prelude.maude`[21]. Listing 2.8 illustrates the `SET{X::TRIV}` module parameterised by the `TRIV` functional theory in Listing 2.10. Conversely, Listing 2.11 shows the `INT` module and Listing 2.9 illustrates the view `Int`, binding `INT` to `TRIV`. As the `TRIV` theory consists of the only sort `Elt`, the `Int` view simply binds the `Int` sort[22] in `INT` to `Elt`, Listing 2.9 Line 2. As a consequence, by binding the module `INT` to `TRIV`, the `Int` sort will be accessible from the parameterized module `SET{X :: TRIV}` by

[20] This thesis requires to define only functional parameterized modules. The reader interested in parameterised system modules can find a thorough discussion in [19], Section 6.3.

[21] Henceforth, modules shipped with Maude v3.0 will be referred to as *native Maude modules* or *Maude library*. Maude v3.0 is a free software project distributed under GNU-GPLv2.0 and available at [49].

[22] This sort defines a Maude integer.

the view sort `X$Elt`. The binding of `INT` to `TRIV` is achieved via the `Int` view by the importation instruction `including SET{Int}`. Graphically, the binding is indicated by the gray solid line in Figure 2.3. As a result, `including SET{Int}` defines a sort `Set{Int}`, supersort of `Int`, represented as `X$Elt` in Listing 2.8 Line 5. Finally, the operator `_,_`, in Line 8, can be used to generate non-empty sets of integers, hence $\{1, 2, 3\}$ is modelled as the term `1, 2, 3` of sort `Set{Int}`.

```
1  fmod SET{X :: TRIV} is
2    protecting EXT-BOOL .
3    protecting NAT .
4    sorts NeSet{X} Set{X} .
5    subsort X$Elt < NeSet{X} < Set{X} .
6
7    op empty : -> Set{X} [ctor] .
8    op _,_ : Set{X} Set{X} -> Set{X}
9             [ctor assoc comm id: empty prec 121
10             format (d r os d)] .
11
12   op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .
13
       ⋮              ⋮              ⋮

15   endfm
```

Listing 2.8: Definition of the `SET{X :: TRIV}` functional module. Extract from `prelude.maude` library.

```
1  view Int from TRIV to INT is
2    sort Elt to Int .
3  endv
```

Listing 2.9: Definition of the `Int` view from `TRIV` to `INT`. Extract from `prelude.maude` library.

```
1  fth TRIV is
2    sort Elt .
3  endfth
```

Listing 2.10: Definition of the `TRIV` functional theory. Extract from `prelude.maude` library.

```
1  fmod INT is
2    protecting NAT .
3    sorts NzInt Int .
4    subsorts NzNat < NzInt Nat < Int .
5
6    op -_ : NzNat -> NzInt
```

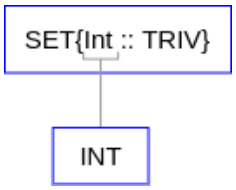


Figure 2.3: Graphical notation to indicate a binding of a functional module to parameter via a theory.

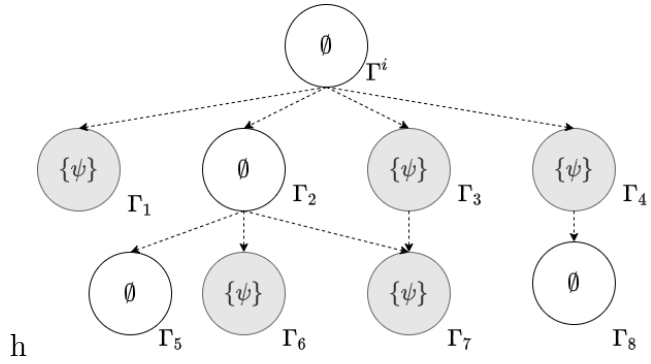


Figure 2.4: Example of a generic reachability analysis.

```
7           [ctor
8            special (id-hook MinusSymbol
9                     op-hook succSymbol (s_ : Nat ~> NzNat)
10                    op-hook minusSymbol (-_ : NzNat ~> Int))] .
11
                ⋮              ⋮              ⋮
13  endfm
```

Listing 2.11: Definition of the `INT` functional module. Extract from `prelude.maude` library.

2.3 Prerequisites for the supported analyses

This section explains the prerequisites to comprehend the analyses supported by the formal verification tool detailed in Sections 3 and 4. Specifically, Section 2.3.1 discusses reachability analysis [6, 10, 19] and the Maude command to perform it. Section 2.3.2 offers a simple definition for the basic Linear Temporal Logic (LTL) [6] formula utilised in Section 5.2.2. Finally, Section 2.3.3 outlines MultiVeStA [48, 53], the Java tool used to perform the statistical analysis in Section 5.3.

2.3.1 Reachability analysis

Reachability analysis is a formal verification technique applicable to transition systems (S, I, R, AP, L) as the one defined in Section 2.1.3, where I consists of the only configuration `Gamma^i`. In fact, the analysis receives two input parameters: an

initial state `Gamma^i` and a condition `cond`. The analysis operates by traversing the entire transition system resulting from the application of all possible rewriting rules on `Gamma^i` and its successors. Finally, the analysis outputs the list of configurations reachable from `Gamma^i` and satisfying `cond`. A graphical example of the analysis is given in Figure 2.4. The illustrated reachability analysis is instantiated with initial configuration Γ^i and condition $AP = \{\psi\}$. The resulting graph represents a transition system whose edges indicate a rewrite rule execution and where each state Γ contains the proposition P s.t. $P \in 2^{AP} \cap L(\Gamma)$. The output set comprises the grey-coloured states, i.e. $\{\Gamma \in S \mid \psi \in L(\Gamma)\} = \{\Gamma_1, \Gamma_3, \Gamma_4, \Gamma_6, \Gamma_7\}$.

In Maude, reachability analysis is conducted by issuing a `search` command. A basic[23] syntax of a `search` is given below.

```
- search in mod : subj searchtype patt such that cond .
```

`mod`	a valid module name;
`subj`	term representing the initial state `Gamma^i0`;
`searchtype`	maximum number of rewriting steps[24] to reach the list of solutions;
`patt`	searched pattern, comprising a variable `v` of sort `S`, denoted as `v:S`;
`cond`	condition which is to be satisfied by the configurations in the output list.

Additionally, it is given a brief description of the command semantics assumed in the rest of this thesis. The `search` command executes a search of the states reachable from the initial term (`subj`) by applying a number of rewrite steps, i.e. consuming a number of `Message`s contained in the initial configuration. The command employs the syntax `patt such that cond` in order to filter the "interesting" states, i.e. those matching `patt` and satisfying `cond` within a number of steps specified by `searchtype`. Those states are returned in an ordered list. Contrarily, in case no reachable state satisfies `cond` or matches `patt`, an empty list is returned.

2.3.2 Linear Temporal Logic

This thesis refers to Linear Temporal Logic (LTL) as an extension of propositional logic complemented with temporal operators: namely, $\Box$ *always*, $\Diamond$ *eventually*, $\bigcirc$ *next* and $\mathbf{U}$ *until*. Since only the $\Box$ operator is required for a sound understanding of the thesis, other operators are omitted[25].

Given the transition system $TS := (S, I, R, AP, L)$, defined in Section 2.1.3, it is provided the semantics of "TS satisfies $\Box\ \phi$", denoted as $TS \models \Box\ \phi$, with $\phi \in AP$ proposition. However, in order to give a meaningful definition, additional notation shall be introduced.

[23] A more extensive explanation can be found in [19], Section 23.

[24] This could be *one* (`=>1`), *one or more* (`=>+`), *zero or more* (`=>*`) and *only terms which cannot be further rewritten* (`=>!`).

[25] The interested reader may want to refer to [6], Chapter 5, for a complete discussion of the LTL syntax and semantics.

A path Π in TS is defined as (2.12).

$$\begin{aligned} \Pi \;:=\; & (\Gamma_0, \Gamma_1, ..., \Gamma_k) \text{ s.t.} \\ & \Gamma_0 \in I \;\wedge \\ & (\Gamma_i, \Gamma_{i+1}) \in R \;\forall\; i \in [0..k) \end{aligned} \tag{2.12}$$

For instance, in Figure 2.4, $(\Gamma^i, \Gamma_3, \Gamma_7)$ is a path. Subsequently, as it is natural, a state $\Gamma \in S$ is said to satisfy an atomic proposition $\phi \in AP$ iff $\phi \in L(\Gamma)$. Additionally, (2.13) expresses the semantics of $\Pi \models \Box\phi$, i.e. "path Π satisfies always ϕ".

$$\Pi \models \Box\, \phi \text{ iff } \Gamma \models \phi \;\forall\; \Gamma \in \Pi \tag{2.13}$$

In Figure 2.4, (Γ_3, Γ_7) is an example of a path satisfying ψ.
Finally, (2.14) defines the semantics of $TS \models \Box\, \phi$.

$$TS \models \Box\, \phi \text{ iff } \Pi \models \Box\, \phi \text{ for all paths } \Pi \text{ in } TS \tag{2.14}$$

Now, it can be observed that the transition system defined in Figure 2.4 does not satisfy $\Box\, \psi$. In fact, no path having Γ^i as first state satisfies $\Box\, \psi$.

A more precise semantics for $TS \models \Box\, \phi$, relying on the definition of the language of atomic propositions satisfying a given LTL formula, can be found in [6], Section 5.1.2.

2.3.3 MultiVeStA: a tool for statistical analyses

MultiVeStA [48, 53, 51] is a Java tool which can be employed to conduct statistical analyses for estimating quantitative properties of discrete-time models. The MultiVeStA approach is based on *confidence interval* (CI) estimation achieved through the execution of a varying number of Monte Carlo simulations [48]. In general, given a simulator of the model, a property to be estimated and a few statistical parameters, MultiVeStA produces a statistically-valid approximation of the quantitative property as displayed by the model. Specifically, in order to better understand the tool functioning, it is essential to examine each input given to MultiVeStA and detail the produced output.

First, the tool expects a discrete-time simulator reproducing the behaviour of the model under analysis. Such a model can be developed in several languages (including Java, R, C++, Python, Maude), although MultiVeStA fully-supports only few of them[26]. In the case the simulator is implemented in a language other than the ones fully-supported, MultiVeStA requires the modeller to perform an additional integration step [48]. That is the implementation of a Java class wrapping the model simulator and extending the class `NewState`. `NewState` is to be viewed as the MultiVeStA interface to the underlying model simulator, allowing the tool to directly control the underlying model transitions and current state. For instance, in the case

[26] Java, R, C++ and Python.

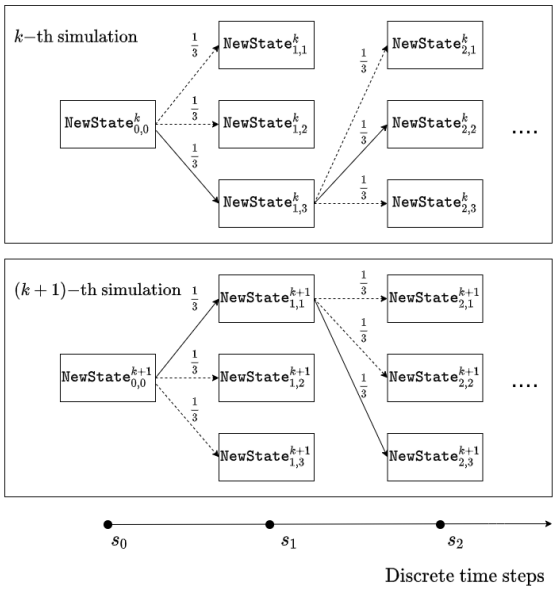


Figure 2.5: Two MultiVeStA execution traces (solid lines), generated by executing a generic simulator. Dashed lines indicate other possible execution scenarios.

of Maude, the `NewState` subclass controls the Maude simulator by starting a Maude interpreter session as an external process[27]. Consequently, the MultiVeStA `NewState` objects contain the current configuration of the underlying model. In other words, they represent the states in the computation graph of the ongoing simulation, as illustrated in Figure 2.5. Besides, the figure illustrates that each simulation generates a different execution trace, as at each step `NewState` uniformly selects the next state. Consequently each simulation can be regarded to as a reproducible[28] Monte Carlo experiment. An essential method of `NewState` is `rval(String obs)`, which, given a property identifier `obs`, returns the value of the queried variable as a real number. In practice, an observed value of a variable is queried in a MultiQuaTEx property and derived from the underlying model configuration, via the `NewState` object.

Second, MultiVeStA properties are expressed in MultiQuaTEx, whose minimal syntax is defined in Listing 2.12[29].

```
1  MQ ::= DS EP
2  EP ::= eval parametric(EL, x1, min, max, s)
3  EL ::= list of E[PExp]
4  DS ::= set of DefOp
5  DefOp ::= N(x1,...,xm) = PExp;
6  SExp ::= c | rval(i) | F(SExp1,...,SExpk) | xj
7  PExp ::= SExp | #N(SExp1, ..., SExpn) |
8           if SExp then PExp1 else PExp2 fi
```

Listing 2.12: MultiQuaTEx syntax, simplified from [48].

[27] By using the ExpectJ library.
[28] To achieve experiments reproducibility, a seeding mechanism is in place.
[29] Refer to [48] for the complete formal syntax.

From the syntax, we observe that a MultiQuaTEx formula is defined as a set of operator definitions (*DefOp*) and a parametric evaluation of a list of expected value observations (*EL*). *DefOp* is an operator definition, comprising the operator signature on left-hand side and a path expression (*PExp*) on the right-hand side. Contrarily, a parametric evaluation is a construct introduced by Sebastio et al. [48], allowing to execute a list of observations (*EL*) for each $x_1 \in [min, max]$, incrementing x_1 by s units. It should be noted that the expected values of the observations (`E`*[PExp]*) are calculated w.r.t. the computed Monte Carlo simulations. Additionally, a path expression, *PExp*, is defined as either a simple state expression *SExp* or recursively as a compound of *PExp*. Specifically, the syntax *#N($SExp_1$, ..., $SExp_n$)* issues the function call *N(...)* in the next state of the current simulation. The statement `if` *g* `then` *x* `else` *y* `fi` has the semantics that if the guard *g* is satisfied in the current state, then the path expression *x* is evaluated, otherwise *y*. Lastly, *SExp* is an expression calculated in a simulation state; it can either be a boolean condition (*c*), a query for the given observation *i* (`rval`*(i)*), an arithmetic or boolean expression (*F($SExp_1$, ..., $SExp_k$)*) or a variable. Listing 4.6 illustrates an example of a MultiQuaTEx property.

Finally, MultiVeStA expects the parameters defining the required confidence interval: the statistical confidence (α) and the CI precision (δ). The usage of these parameters is explained by the following example. Given a model variable X, whose unknown value is x, MultiVeStA approximates x with $\hat{x}$. The resulting approximation $\hat{x}$ is computed by executing a number of simulations n large enough that $\hat{x} \in [x - \frac{\delta}{2}, x + \frac{\delta}{2}]$ with probability $1 - \alpha$. It should be noted, that this happens for a varying, even though frequently large n. This is due to the fact that the distance between the current approximation and the actual value is estimated as $\hat{d}$ (2.15) [53].

$$\hat{d} = k \cdot \sqrt{\frac{\hat{s}^2}{n}} \tag{2.15}$$

In (2.15), k is regarded to as a constant, n as the number of simulations and $\hat{s}$ as the sample variance of X value over the n observations. Consequently, we observe that $\hat{d}$ decreases at the speed of the square root function and $\hat{x}$ is accepted, only when $\hat{d} \leq \frac{\delta}{2}$.

2.4 Notions on stock market modelling

This section introduces basic notions of stock market modelling based on historical data. Section 2.4.1 offers an intuitive understanding of the *geometric Brownian motion* as the means to achieve a feasible predictive model based on past stock market trends.

2.4.1 The geometric Brownian motion

The geometric Brownian motion (GBM) is the continuous-time stochastic process defined in (2.16).

$$P_t = P_0 \cdot exp\left[\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right] \tag{2.16}$$

In the equation, the two constants μ and σ are respectively called *drift* and *volatility*, whereas W_t is a random variable following a *Weiner process*.

$$W_t := \epsilon\sqrt{dt} \tag{2.17}$$

Formally, a Weiner process W_t is defined as the process (2.17) satisfying the following properties:

1. $\epsilon \sim N(0,1)$[30];
2. for any given pair (t_0, t_0'), W_{t_0} and $W_{t_0'}$ are independent.

In other words, a W_t is the component yielding the stochastic behaviour of a GBM. Additionally, the geometric Brownian motion as a whole can be viewed as the harmonic result of two components [31]:

1. the drift component $\left(\mu - \frac{\sigma^2}{2}\right)t$;
2. the volatility component σW_t.

The effects of the two aforementioned components on the resulting process is shown in Figure 2.6.

As it is evident from the figure, the drift component defines the trend of the resulting process, whereas the volatility component is a measure of the randomly sampled shocks. Intuitively, this signifies that negative values for μ yield to a downward prediction trend, whereas positive ones to a growth. Oppositely, the higher the σ is, the more significantly the prices predictions change.

In the literature [21, 31], the two constants μ and σ are estimated based on the daily log returns of the targeted stock market. Given the closing prices of two consecutive trading days C_1 and C_2, the log return w.r.t. the second trading day is defined as $ln(C_2) - ln(C_1)$.

[30] It denotes that ϵ follows a standard normal distribution.

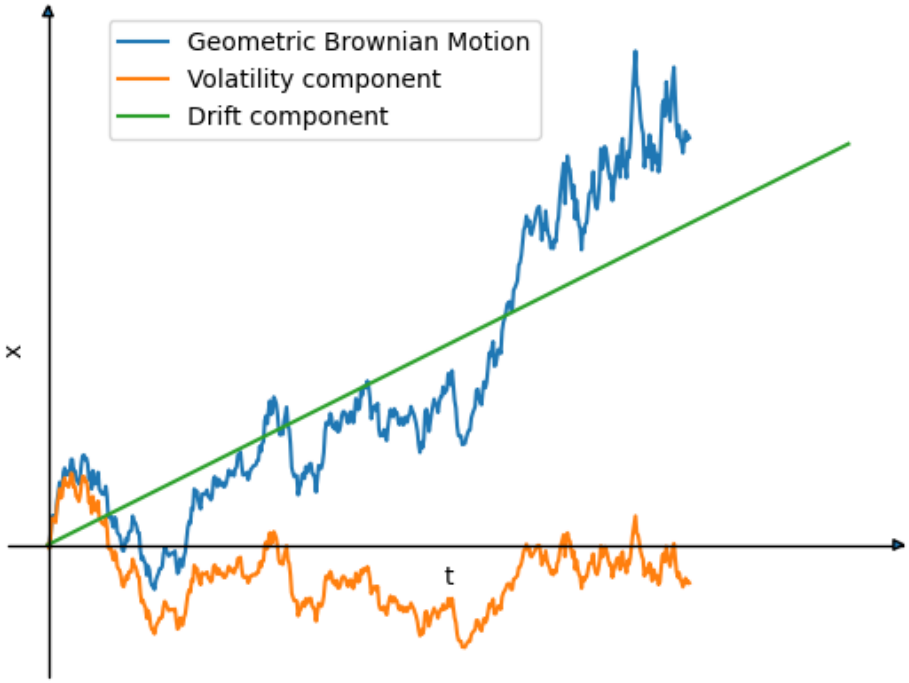


Figure 2.6: The geometric Brownian motion components.

3 A Maude specification of Lending Pools

This chapter discusses a Maude specification of the lending pools model presented in Section 2.1.3. This specification will allow to automatically simulate the LP rules on arbitrary configurations. Subsequently, in Section 4, this development will be extended in order to conduct statistical analysis on LP.

Specifically, Section 3.1 describes the basic types required to model LP configurations. Section 3.2 outlines the implementation of the model rules. Finally, Section 3.3 discusses the language features used for the model parameterisation and testing.

As reminded in each listing throughout this chapter, Appendix A contains the essential modules of the developed specification. The full specification is open source, under the GNU General Public License version 2 at [38].

3.1 The model basic components

This section describes the Maude modules developed in order to represent the basic components of an LP configuration. Section 3.1.1 overviews the tokens type system and introduces the abstraction implementing mathematical functions. Section 3.1.2 exemplifies the usage of such abstraction, defining an essential component of an LP configuration: a pool.

3.1.1 Tokens

The model formalised by Bartoletti et al. [7] heavily relies on basic mathematical concepts including sets and functions, mainly in order to formalise the mechanisms used for transferring tokens among parties. Consequently, the tokens abstraction is used by this section to explain how these mathematical means, namely sets and functions, have been defined in Maude. In order to achieve this, the section discusses how the following concepts, defined in Section 2.1.3, have been specified:

1. the basic token types, minted and free tokens;
2. the sets of token types, $T_f \subseteq \mathcal{T}_f$ and $T_m \subseteq \mathcal{T}_m$;
3. the functions from token types to non-negative real numbers, σ_A, for agent A.

Distinguishing between free and minted tokens is essential in order to specify LP. Precisely, it is desirable to model τ_i and τ_i' $\forall\ i \in \mathbb{N}$ as respectively free and minted tokens. Listing 3.1 shows how this is obtained in Maude. For instance, the operators `tau(_)` and `_'`, in Lines 6 and 7, allow to specify τ_0 with the term `tau(0)` and τ_0' with `tau(0)'`, respectively of sort `Free-Token` and `Minted-Token`. Notably, free and minted tokens are defined as subsorts, `Free-Token`, `Minted-Token`, of `Token` (Lines 3 and 4).

```
1  fmod TOKENS is
       ⋮              ⋮              ⋮
```

```
3       sort Token Free-Token Minted-Token .
4       subsort Minted-Token Free-Token < Token .
        ⋮                ⋮                ⋮

6       op tau(_) : Nat -> Free-Token [ctor ...] .
7       op _' : Free-Token -> Minted-Token [ctor ...] .
        ⋮                ⋮                ⋮

9   endfm
```

Listing 3.1: Maude definition of `Free-Token` and `Minted-Token`, extract from Listing A.3, Line 8

In order to manage several free and minted tokens, two types for sets of token types have been developed: `Set{Free-Token}` and `Set{Minted-Token}`. For example, the two types allow to specify the set of free tokens $\{\tau_0, \tau_1, \tau_2\}$ as the term `tau(0), tau(1), tau(2)` of sort `Set{Free-Token}`. Conversely, the set of minted tokens $\{\tau_0', \tau_1', \tau_2'\}$ is expressible as the term `tau(0)',tau(1)', tau(2)'` of sort `Set{Minted-Token}`. The parametric module `SET{X::TRIV}`[31] is used to implement `Set{Free-Token}` and `Set{Minted-Token}`, representing free and minted tokens sets, respectively. Similarly to the relation we have observed in Listing 3.1, also `Set{Free-Token}` and `Set{Minted-Token}` are defined as subsorts of `Set{Token}`. This is achieved by the parameterised module `SUBSET{X::TRIV,Y::TRIV}`[32] in Listing A.3, Line 44, which allows to define "mixed" token type sets such as $\{\tau_0, \tau_1, \tau_1'\}$ as `tau(0),tau(1), tau(1)':Set{Token}`.

Functions are the last abstraction which serves to model essential LP objects, including agents' wallets, π_f, π_m and π_l. Consequently, it is desirable to define functions, as per their usual mathematical definition[33], in a parametric manner. In this sense, the sorts `Map{X,Y}`, `Set{X}` and `Set{Y}` are employed: `Map{X,Y}` can be used to store the pairs composing a function, as shown in Section 2.2.1, while `Set{X}` and `Set{Y}` for the definition of the operators computing the function domain and codomain. Listing 3.2 shows the approach used to define a generic function offering the operations for determining its domain (`dom`) and codomain (`cod`[34]). Notably, the `FUNCTION` module is parameterised with respect to a `MAP-THEORY`, `MT`, acting as the entity binding `Map{X,Y}` with the sorts corresponding to its domain (`Set{X}`) and codomain (`Set{Y}`). The operator `dom` is defined via the operator `$dom` which utilises all the sorts and operators of `MT`, in order to produce a set of `MT$D`-sorted terms. Listing 3.3 illustrates the sorts and operators of a `MAP-THEORY`, described below.

`E`, `M` - represent respectively an `Entry{X,Y}` and a `Map{X,Y}`;

[31] The parameterised module `SET{X::TRIV}` and the view `TRIV` are defined in Listings 2.8 and 2.10, respectively.

[32] Listing A.2, Line 101.

[33] A function f of domain D and codomain C, denoted as $f : D \to C$, is defined as $f \subseteq D \times C$ and $\forall c \in C \; \exists! \; d \in D \; . \; (d, c) \in f$.

[34] For brevity, only the implementation of `dom` is illustrated, as `cod` is analogous.

`D`, `DT` - define the domain type (in the form of `Set{X}`) and its elements' type (`X`);

`C`, `CT` - model the codomain type (in the form of `Set{Y}`) and its elements' type (`Y`).

Similarly, the operators in `MAP-THEORY` are defined as the main operators in `MAP{X::TRIV,Y::TRIV}` and `SET{X::TRIV}`, as further explained.

`_|->_` - is syntactically equivalent to the constructor for a term of sort `Entry{X,Y}`, `_|->_`, Listing 2.1 Line 14.

`_;_` - is syntactically equivalent to the constructor for a term of sort `[Map{X,Y}]`, `_;_`, Listing 2.1 Line 15.

`_,_` - is syntactically equivalent to the constructor of non-empty `Set{X}` in Listing 2.8 Line 8.

```
1   fmod FUNCTION{MT :: MAP-THEORY} is
2       var map : MT$M .
3       var domElt : MT$DT .
4       var codElt : MT$CT .
5       var domSet : MT$D .
6       var codSet : MT$C .
7
8       --- dom implements the function domain as a set
9       --- of MT$D-sorted terms
10      op dom : MT$M -> MT$D .
11      eq dom(map) = $dom(map, emptyD) .
12
13      --- $dom is dom auxilary function, implementing
14      --- the actual logic to iterate over the map and
15      --- extract its values
16      op $dom : MT$M MT$D -> MT$D .
17      eq $dom((map ; domElt |-> codElt), domSet) =
18                                  $dom(map, (domSet, domElt)) .
19      eq $dom((domElt |-> codElt), domSet) = domSet, domElt .
20      eq $dom(emptyM, domSet) = domSet .
21
22      --- cod implements the function codomain as a set
23      --- of MT$C-sorted terms
24      op cod : MT$M -> MT$C .
        ⋮              ⋮              ⋮

26  endfm
27
```

Listing 3.2: Maude definition of the `FUNCTION` main functionalities: `dom` and `cod`, extract from Listing A.2, Line 285

```
1  fth MAP-THEORY is
2      sorts M E DT CT D C .
3
4      --- entry (pair) is subsort of map (set of pairs)
5      subsort E < M .
6      --- constructing the map pairs
7      op _|->_ : DT CT -> E [ctor] .
8      --- ‘assoc‘ and ‘comm‘ make map pairs unordered
9      op _;_ : [M] E -> [M] [ctor id: emptyM assoc comm prec
10                            121 format (d r os d)] .
11     --- empty pair
12     op emptyM : -> M .
13
14     --- CT is the sort of the terms in Set{C}
15     subsort CT < C .
16     --- DT is the sort of the terms in Set{D}
17     subsort DT < D .
18
19     --- operator to be bound to (empty).Set{C}
20     op emptyC : -> C .
21     --- operator to be bound to (_,_).Set{C}
22     op _,_ : C CT -> C
23     [ctor assoc comm prec 121 format (d r os d)] .
24
25     --- operator to be bound to (empty).Set{D}
26     op emptyD : -> D .
27     --- operator to be bound to (_,_).Set{D}
28     op _,_ : D DT -> D
29     [ctor assoc comm prec 121 format (d r os d)] .
30 endfth
```

Listing 3.3: Maude definition of `MAP-THEORY`, binding `Map{X,Y}` with `Set{X}` and `Set{Y}`, extract from Listing A.2, Line 229

As exemplified in Section 3.1.2, this approach allows the modeller to instantiate a function by reusing the logic in Listing 3.2 to compute its domain and codomain, by module parameterisation[35]. This is achieved following the procedure[36]:

1. defining a generic module `<FUN-MOD>` containing the sorts and operators to be bound to `MAP-THEORY` ones;
2. producing a generic view `<FUN-VIEW>` binding `<FUN-MOD>` to `MAP-THEORY`;
3. instantiating the function by `including FUNCTION{<FUN-VIEW>} .`

As a result, the module importing `FUNCTION{<FUN-VIEW>}` is equipped with sort `Map{X,Y}` and the operators `dom` and `cod` returning respectively a `Set{X}`-typed domain and a `Set{Y}`-typed codomain.

[35] Section 2.2.4 outlines Maude parametric programming.
[36] More extensively described in [19], Section 6.3.4.

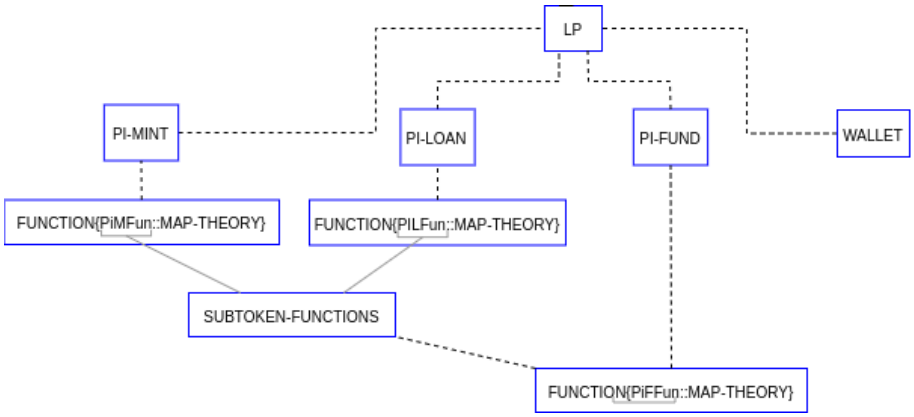


Figure 3.1: Importation graph of the `LP` functional module

3.1.2 Pools

The Maude implementation of a pool, π, defined in Section 2.1.3, is based on `FUNCTION{MT::MAP-THEORY}`, as shown in Figure 3.1. The figure depicts the modules imported in `LP`, which implement the main functionalities of π. The pool components (functions π_f, π_l and π_m) are modelled via `FUNCTION{MT}` and instantiated by means of the views `PiFFun`, `PiLFun` and `PiMFun` binding the specific `FUNCTION{MT}` operators and sorts to `MAP-THEORY` ones. The remaining included module, namely `WALLET`, defines basic operations on wallets which are useful to specify π.

In order to clarify the usage of `FUNCTION`, the rest of the section describes `PI-FUND`, the module containing the specification for π_f[37]. Listing 3.4 illustrates how to instantiate `FUNCTION{MT}` by the view `PiFFun`.

```
1  fmod PI-FUND is
2      protecting FUNCTION{PiFFun} .
       ⋮              ⋮              ⋮
4  endfm
```

Listing 3.4: Maude definition of `PI-FUND`, extract from Listing A.5, Line 38

Listing 3.5 fully specifies the way the binding, operated by the view, is achieved. Specifically, the `Map` underlying π_f[38] (`MT$M`, in `FUNCTION{M}`, Listing A.2, Line 286) is a `Map` from a `Token` to a `Float0+`[39]. As a result, the basic sorts for domain and codomain elements, `DT` and `CT`, are bound to `Token` and `Float0+`. Similarly, the domain and codomain sorts, `D` and `C`, are bound to `Set{Token}` and `Set{Float0+}`, respectively. Additionally, it is worth observing that `FUNCTION{PiFFun}` not only offers

[37] `PI-LOAN` and `PI-MINT` are developed similarly so to model the remaining pool components: π_l and π_m.

[38] Defined in the original formal model as a function mapping a free token type to a rational number.

[39] A real non-negative number.

the means to model wallets[40], but it is also fundamental for defining `FUNCTION{PiLFun}`[41] and `FUNCTION{PiMFun}`. In fact, as shown in Figure 3.1, `FUNCTION{PiFFun}` is included in `SUBTOKEN-FUNCTIONS`, the module representing the sorts and operators bound by `PiLFun` and `PiMFun`.

```
1  view PiFFun from MAP-THEORY
2                   to TOKEN-MAP is
3      sort M to Map{Token, Float0+} .
4      sort E to Entry{Token, Float0+} .
5
6      sort DT to Token .
7      sort CT to Float0+ .
8
9      sort D to Set{Token} .
10     sort C to Set{Float0+} .
11
12     op emptyD to empty .
13     op emptyC to empty .
14 endv
```

Listing 3.5: Maude definition of `PiFFun`, the view binding `MAP-THEORY` to the sorts and operators specifying π_f, extract from Listing A.5, Line 106

3.2 LP rules

This chapter draws the attention on specification of the LP actions in Maude. First, Section 3.2.1 defines the LP configurations modelled in Maude. Second, Section 3.2.2 examines the semantics of the object-based approach and motivates the main changes to the initial model notation.

3.2.1 Configurations definition

Listing 3.6 expresses an example of a simple LP configuration in Maude.

```
[
  Pi | p
  (< A : noState | * sigma_A >
   < B : noState | * sigma_B >
   < C : noState | * sigma_C >)
   < R(n) : Round | none >
   < P(0) : LiqParams | CMin(cminV),
                        Rliq(rliqV) >
] :: ClosedConfiguration
```

Listing 3.6: Maude definition of a simple LP configuration

The Maude representation of a (`Message`-less[42]) LP configuration comprises nearly

[40] As defined in Section 2.1.3.
[41] Note that its codomain can be modelled as a `Map{Token, Float0+}`.
[42] Section 3.2.2 discusses LP configuration holding messages.

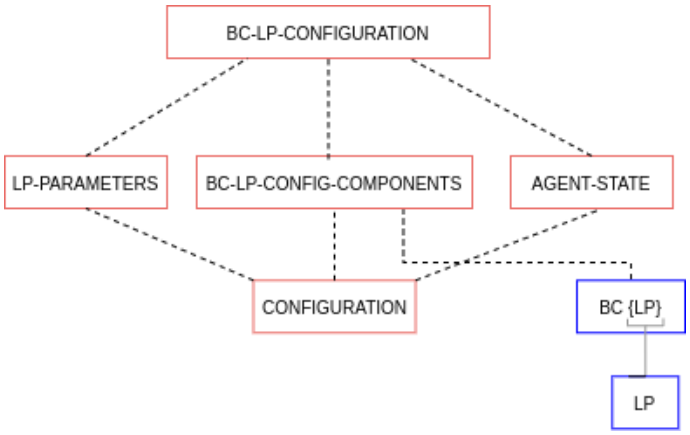


Figure 3.2: Importation graph of the `BC-LP-CONFIGURATION` system module

the same components, as discussed in Section 2.1.3. Firstly, `Pi | p` is a term of sort `BC{LP}`, defined as an `Object` in Listing A.7, Line 58. The term `pi` models the pool π state. Conversely, `p` models the assets price function, p, via a simple `Map{Token, Float0+}`. Secondly, a generic object of the form `< Id : noState | * sigma_Id >` represents an agent identified by `Id`, having a `noState` behaviour[43] and holding a wallet `sigma_Id`[44]. Thirdly, the term `< R(n) : Round | none >` is an `Object` added to the configuration in order to store the `n`-th blockchain round. Finally, `< P(0) : LiqParams | CMin(cminV), Rliq(rliqV) >` is employed to parameterise the configuration given by `CMin` (`cminV`) and `Rliq`[45]. (`rliqV`) parameters, as shown in Section 3.3.1. Additionally, it is worth noting that the sort of a LP configuration term is `ClosedConfiguration`. A `ClosedConfiguration` is obtained from a `Configuration` by the operator `[_] : Configuration -> ClosedConfiguration`, as shown in Listing 3.6. As Section 3.2.2 shows, this is a fundamental operator in order to modify only specific components of a `Configuration`, while maintaining the rest intact.

The full implementation of LP configuration is contained in `BC-LP-CONFIGURATION`, whose main dependencies are pictured in Figure 3.2. Specifically, `BC{LP}` contains the sorts and operators discussed in Section 3.1.2, whereas `CONFIGURATION` is the essential system module explained in Section 2.2.2. Next, `AGENT-STATE` and `BC-LP-CONFIG-COMPONENTS` define additional syntactic sugar, mainly destructuring operators, which allow to decompose a `Configuration` into its atomic components. Lastly, `LP-PARAMETERS` defines the lending pools essential parameters discussed in Section 3.3.1.

3.2.2 Translating LP actions into Maude rules

As stated in Section 3.2.1, the objects abstraction is essential for defining LP actions. Specifically, `Objects` (in case of lending pools, the agents and the pool) need to

[43] Section 4 details a specific behaviour type.
[44] Wallet of agent identified by Id, σ_{Id} in Section 3.1.2.
[45] Namely, $C_{\min}$ and r_{liq}.

interact with one another by means of `Messages`. In these specifications, LP actions are syntactically modelled as `Messages`, whereas their semantics are specified by the rewriting rules consuming those `Messages`. As a result, the LP rule $\mathrm{r}_A(z^n)$ of a given arity n is syntactically defined as the Maude operator in Listing 3.7.

```
op r : Agent-Id S1... Sn -> Message .
```

Listing 3.7: Syntax of a message representing an issued LP actions in Maude.

In the listing, `S1...Sn` are the sorts of the terms `t1...tn` representing the n input parameters z^n. Contrarily, the first input parameter of the operator `r` is a term `A:Agent-Id`, identifying the agent issuing the message. Consequently, the action $\mathrm{r_A}(z^n)$ is represented in Maude by `r(A, t1, ..., tn):Message`.

However, actions cannot be implemented as mere syntactic constructs, since they directly imply a system state change, whose semantics ought to be defined. This is exactly the purpose of the rewriting rules subsequently discussed. Specifically, conditional rewriting rules, following the syntax in Listing 3.8, are used to model LP actions semantics.

```
crl [r-id] :
    [ gamma r(A:Agent-Id, t1:S1, ..., tn:Sn) ]
                                              => [ gamma' ]
          if G(gamma, A, t1, ..., tn) .
```

Listing 3.8: A generic Maude rewrite rule expressing the semantics of an LP action.

Here, `[r-id]` is the rule identifier, mainly used for debugging purposes, whereas `gamma` and `gamma'` are terms of sort `Configuration`, representing respectively the system state before[46] and after the rewrite occurs. Next, `r(A, t1, ..., tn)` is the message whose consumption entails the transition from state `gamma` to `gamma'` according to the semantics expressed by `gamma'` and `G(gamma, A, t1, ..., tn)`. Lastly, `G(gamma, A, t1, ..., tn)` (also named a guard) is a proposition, which given the left-hand side terms, allows the rule to be executed only if it is evaluated to true. Thus, intuitively, the aforementioned rule can be executed in a state `[ gamma0 ]` if and only if `gamma0` contains `r(A, t1, ..., tn):Message`[47] and `G(gamma0, A, t1, ..., tn)` is true. Noteworthily, the implementation choice of the `ClosedConfiguration` constructor (`[_]`) is essential to match the unmodified `Objects` of `gamma0` against the rule left-hand side.

As a result, a rewriting rule implementing the LP action $\mathrm{r}_A(z^n)$ induces the consumption of the message `r(A, t1, ..., tn)`. Clearly, the rewrite application yields changes to the initial configuration according to the semantics of the LP action. The rest of this section exemplifies this point, by describing the liquidate LP action $\mathrm{Liq}_A(B, v : \hat{\tau}, \tau')$, as defined in Figure 2.1.

[46] Modulo `r(A, t1, ..., tn)`.

[47] For some `A, t1, ..., tn` terms of sorts `Agent-Id, S0, ..., Sn`

```
1   crl [liquidate] :           --- i. represents the i-th
2      [ gamma                  --- constraint in the liquidate
3      < R(r) : Round | none > --- action def (Figure 2.1)
4      < A : agState | * sigmaA >
5      < B : agState' | * sigmaB >
6      (pi | p)
7      liquidate(A, B, (v, hTau), tau') ]
8      => [ gamma
9         < R(r + 1) : Round | none >
10        < A : agState | * sigmaA' >
11        < B : agState' | * sigmaB' >
12        (pi' | p) ]
13     if v' := v * (p[hTau] / (p[(pi).u(tau')])) *      --- 4.
14                                  (gamma).Rliq /\
15        sigmaA' := (sigmaA - v : hTau) + v' : tau' /\ --- 8.
16        sigmaB' := sigmaB - v' : tau' /\              --- 9.
17        (gamma  < A : agState | * sigmaA >
18                < B : agState' | * sigmaB >
19                (pi | p) ).C(B) < (gamma).CMin /\    --- 10.
20        gamma' := gamma                    --- final config
21              < A : agState | * sigmaA' >  --- after rule
22              < B : agState' | * sigmaB' > --- application
23              (pi' | p) /\
24        (gamma').C(B) <= (gamma).CMin .              --- 11.
```

Listing 3.9: Definition of $\mathrm{Liq}_A(B, v : \hat{\tau}, \tau')$ in Maude, simplified extract from Listing A.7, Line 1188. Apart from the removed rules, the only modification to the appendix is the renaming of `tau` in `hTau`. This is to align the notation with Figure 2.1.

In the liquidate rule, the left-hand side matches (i.e. it can be applied to) any configuration containing a liquidate message involving two existing users and satisfying the rule condition. The condition is a simplification of the original, showing the action semantics of the essential constraints[48] detailed in Section 2.1.3. Initially, it should be noted that only the components destructured from the left-hand side `Configuration` are modified by the action: liquidator (`A`) and borrower (`B`) agents[49], blockchain round object and `(pi | p)`. Obviously, the input terms to the liquidate message[50] are utilised as well. Subsequently, the rule condition shows how these parameters are used, explaining the exact action behaviour. First, matching equation having `v'` as the left-hand side assigns to `v'` the units of `tau'`, i.e. the amount of the liquidation seized collateral. Second, the liquidator and borrower's initial balances are updated reflecting the fact that the liquidator repays `v` units of `hTau` (borrower's loan) in return for `v'` units of `tau'`. Third, it is verified that the collateralization of `B` in the left-hand side configuration is below `CMin`[51]. Lastly, the final `ClosedConfiguration`

[48] The preconditions (4), (8), (9), (10) and (11) in Figure 2.1.

[49] Henceforth, agent `Objects` will be referred to by their id term of sort `Oid`.

[50] `A, B, (v, hTau), tau'`

[51] The configuration destructors of `CMin` (`(gamma).CMin`) and of the other LP parameters are discussed in Section 3.3.1.

(`gamma'`) is reconstructed from the updated objects[52] and `B`'s collateralization, based on `gamma'`, is compared to `CMin`. Thus, the condition is satisfied, i.e. the rule is executable, if and only if the borrower's collateralization is less or equal than `CMin`, after the action has been executed[53].

3.3 The model specifications

This section summarises the current specifications development, underlining two particularly desirable properties expected from any high-quality piece of software: parameterization capability and testability. Specifically, Section 3.3.1 discusses the parameters which could be used to instantiate an LP configuration, whereas Section 3.3.2 focusses on the Maude features employed for testing the specifications.

3.3.1 The model parameterization

The lending pools model ought to have a number of parameters in order to offer a more realistic representation of the underlying financial platforms. Consequently, the Maude specification of LP should expose functionalities in order to set and modify those parameters. The model can be instantiated by choosing at least three parameters (C_{min}, r_{liq} and *Maxliq*), introduced in Section 2.1.3. The respective terms specifying them in Maude are presented below.

`CMin` - the threshold indicating the minimum collateralization a user may have without risking to be liquidated. It models C_{min}.

`Rliq` - the discount rate applied to the amount of tokens seized by the liquidator during a liquidation. It models r_{liq}.

`Maxliq` - threshold expressing the maximum amount of repayable loan during a single liquidation. It models *Maxliq*.

These parameters are defined in `LP-PARAMETERS`[54], and the value of `CMin` and `Rliq` can be modified via the operator `replaceLiqParams`, in Listing 3.10 Line 20. Notably, the operator either inserts a new `Object` of class `LiqParams`[55] (Line 30) or it updates the current `CMin` and `Rliq` values (Line 32). Therefore, reducing the term `(gamma).replaceLiqParams(1.5, 1.1)` determines that `CMin` is set to `1.5` and `Rliq` to `1.1`. The new values are then accessed through `gamma`'s destructors defined in Listing 3.11. Examples of their usage can be observed in Listing 3.9 Lines 14 and 19 for `Rliq` and `CMin`, correspondingly.

```
1  mod BC-LP-MODEL-MODIFIERS is
       ⋮              ⋮              ⋮
```

[52] Namely `A`, `B` and `pi'`, whose update is not shown in Listing 3.9
[53] I.e. when the system is in state `gamma'`.
[54] Listing A.7, Line 111
[55] As shown in Listing 3.6

```
3   --- operator used to insert a new ‘LiqParams‘ object
4   op addLiqParams(_,_,_) : Configuration Float0+
5                            Float0+ ->
6                               Configuration .
7   --- equation inserting the new object
8   eq addLiqParams(gamma, cminV, rliqV) =
9         gamma < P(0) : LiqParams | CMin(cminV),
10                                   Rliq(rliqV) > .
11  --- operator wrapping the ‘Configuration‘ into a
12  --- ‘closedConfiguration‘
13  op addLiqParams(_,_,_) : closedConfiguration Float0+
14                           Float0+ ->
15                                 closedConfiguration .
16  eq addLiqParams(cGamma, cminV, rliqV) =
17       [ addLiqParams((cGamma).config, cminV, rliqV) ] .
18
19  --- operator updating ‘CMin‘ and ‘Rliq‘ values
20  op (_).replaceLiqParams(_,_) : closedConfiguration
21                                 Float0+ Float0+
22                                 -> closedConfiguration .
23  eq (cGamma).replaceLiqParams(cminV, rliqV) =
24     [ ((cGamma).config).$replaceLiqParams(none,
25                                           cminV, rliqV) ] .
26  op (_).$replaceLiqParams(_,_,_) : Configuration
27                                    Configuration
28                                    Float0+ Float0+
29                                      -> Configuration .
30  eq (none).$replaceLiqParams(gamma', cminV, rliqV) =
31            addLiqParams(gamma', cminV, rliqV) .
32  eq (gamma < P(0) : LiqParams | attrS >)
33               .$replaceLiqParams(gamma', cminV, rliqV) =
34  gamma < P(0) : LiqParams | CMin(cminV),
35                             Rliq(rliqV) > gamma' .
      ⋮                ⋮                ⋮

37  endfm
```

Listing 3.10: Maude definition of the operator `replaceLiqParams`, allowing to modify the LP parameters. Extract from Listing A.7, Line 1793

```
1   mod BC-LP-CONFIGURATION is
      ⋮                ⋮                ⋮

3   --- assumes there is only one LiqParams object in gamma
4   op (_).params : Configuration -> [Configuration] .
5   eq (gamma).params = $params(gamma) .
6   --- The auxiliary operator $params, returning the LiqParams
7   --- object in gamma, is omitted
      ⋮                ⋮                ⋮
```

```
9
10  op (_).CMin : Configuration -> Float0+ .
11  eq (gamma).CMin = ((gamma).params).CMin .
12
13  op (_).Rliq : Configuration -> Float0+ .
14  eq (gamma).Rliq = ((gamma).params).Rliq .
      ⋮              ⋮              ⋮

16  endfm
```

Listing 3.11: Maude definition of the operators used to access `CMin` and `Rliq` values in a given configuration `gamma`. Extract from Listing A.7, Line 330

3.3.2 Validation of the LP specifications

The standard Maude command `reduce` is employed to unit-test the specifications. The `reduce` command (abbreviated by `red`) reduces input terms into their corresponding canonical form terms, by progressively applying equational simplification steps, as detailed in Section 2.2.1. The deterministic behaviour of reduce makes it a suitable command to test the specifications. In order to make the tests results immediately verifiable, the test cases were designed so that they would always return a boolean literal that equals to true if and only if the test case passes. Listing 3.12 shows an example of a set of unit tests for a simple operator defined over agents' wallets, their semantics is indicated in the comments above each test.

```
--- If the token type of the added value is not in the map
--- -> a new entry is added
red in WALLET : (tau(0) |-> 1.0) + 5.0 : tau(1)  ==
                ((tau(0) |-> 1.0) ; (tau(1) |-> 5.0)) .

--- If the token type in map -> token amount is summed up
red in WALLET : (tau(0) |-> 1.0) + 5.0 : tau(0)  ==
                (tau(0) |-> 6.0) .

--- If the map comprises multiple entries -> sum the
--- the value of the right token type
red in WALLET : ((tau(0) |-> 1.0) ; (tau(1) |-> 1.0)) +
                              5.0 : tau(0) ==
                 (tau(0) |-> 6.0) ; (tau(1) |-> 1.0) .
```

Listing 3.12: Unit-test of the operator summing a `Token` to `Float0+` with a `Token Float0+` pair, as defined in Listing A.4, Line 9.

Conversely, the more advanced `search` command is utilised to perform "integration-tests" on the rewriting rules of the specifications. In fact, `search` is capable of exploring the entire state space reachable by applying a finite number of rewriting rules, as explained in Section 2.3.1. These test cases could not be designed so to return boolean values, as they test properties on multiple configurations. Thus, they

appear less immediate to verify than the aforementioned tests. However, they are still very convenient to test interactions between the model components. Listing 3.13 illustrates an integration test for the rule modelling the heuristic which powers rational liquidators, discussed in Section 4.

Totally, more than 350 tests have been developed in order to validate the specification of the LP model. For brevity, tests have not been included in the Appendices. However, they are available in the project repository [38].

```
--- This checks the number of all canonical final states,
--- which is 7.
search in LIQUIDATOR-TEST :
   (test12) =>! X:closedConfiguration .
--- Number of canonical final states satisfying the where
--- LED(0)'s collateralization is 0.0, then
--- expected = 7
search in LIQUIDATOR-TEST :
   (test12) =>! X:closedConfiguration
   such that
        ((X:closedConfiguration).config).C(LED(0)) == 0.0  .
```

Listing 3.13: Integration test of the liquidator agent behaviour, in scenario when borrower collateralization is less than `Rliq`. The tested rule is defined in Listing A.8, Line 395.

4 An LP simulator for liquidating agents

This chapter lays the foundations for tackling the significant research problem of finding optimal `CMin` and `Rliq` parameters for the lending pools model. This is achieved by instantiating an LP simulator for conducting statistical analyses of the model. The simulator comprises:

1. the LP Maude specification, defined in Section 3;
2. a heuristic for automating the behaviour of rational liquidators, defined in Section 4.1;
3. a price model instantiated for simulating three different prices evolution scenarios, for the three most widely employed cryptocurrencies, defined in Section 4.2;

Finally, the integration of the resulting LP simulator with MultiVeStA, is presented in Section 4.3.

4.1 A fully-automated liquidating heuristic

This section explains a liquidating heuristic causing the LP protocol to possibly reach unsafe states, where loans are not guaranteed to be repaid. Section 4.1.1 offers an intuitive understanding of aggressive liquidating behaviours. Subsequently, Section 4.1.2 describes the proposed liquidating algorithm and Section 4.1.3 discusses its implementation.

4.1.1 The impact of liquidations on collateralization

As observed in Section 2.1.3, liquidate actions involve two agents, namely a liquidator and an undercollaterized borrower. Specifically, the liquidator is regarded to as an agent holding sufficient liquidity to issue liquidate actions, whereas an undercollaterized borrower is an agent having its collateralization below `CMin`, the liquidation threshold.

Practically, liquidators have a fundamental role in the LP financial safety, as they are the actors supplying free tokens whenever the pool is lacking them. Although they enable an essential mechanism for the LP safety[56], excessively tenacious liquidators could be harmful to the system. Specifically, these behaviours could disincentivize the liquidated agents (i.e. the undercollaterized borrowers[57]) to ultimately repay their loans. This is better understood by observing Figure 4.1, where all the liquidating scenarios are outlined. The figure illustrates the agents' collateralization[58], detailing the outcomes of liquidate actions in every possible (non-trivial) configuration. The scenarios are also well captured by the running example in Figure 2.2, used to illustrate the liquidate action.

[56] Together with interest rates.
[57] Henceforth simply borrowers.
[58] Defined in Section 2.1.3.

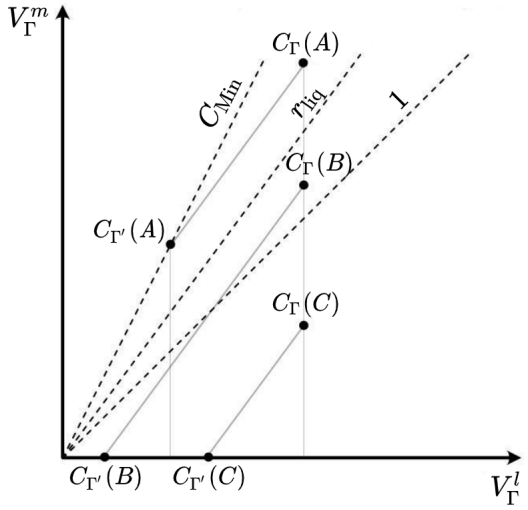


Figure 4.1: The three possible liquidation scenarios

Firstly, the three dashed lines in the figure correspond to the liquidation parameters specific to the instantiated pool. Their labels represent the respective line slopes. Specifically, C_{min} and r_{liq} were discussed in Section 3.3.1, whereas the line labelled 1 depicts the scenarios where the collateral value equals the loan value[59]. Consequently, it can be intended as the loan repayment incentivizing threshold, i.e. the collateralization value below which agents should be considered to be disincentivized in repaying their loans. These residual loans are also called *non-recoverable*.

Additionally, the three points indicate the initial collateralization of three liquidated borrowers. Each liquidation action[60] is illustrated by a solid line drawn from $C_\Gamma(I)$ to $C_{\Gamma'}(I)$ for $I \in \{A, B, C\}$. Liquidations entail a decrease in the liquidated user's collateralization by a linear factor proportional to r_{liq} and ultimately determined by the liquidator. It should be noted that the liquidation actions described in the figure follow the exact semantics of the liquidate as defined in Listing 3.9, as the resulting loan value must be greater than zero[61] and the final collateralization must be at most C_{min}[62].

Lastly, it is worth observing that the liquidations in the figure can be achieved by applying only one action if and only if two conditions hold. Firstly, the liquidator invests enough liquidity to seize the entire seizable collateral. Secondly, the liquidated borrower does not diversify the type of the loan. If either the first condition or the second does not hold, then the liquidations illustrated in the figure can be achieved uniquely by performing several liquidate actions on the borrower. This is frequently the case in the major LP implementations: Compound and Aave. In fact, these

[59] Given a borrower B, collateral value and loan values are defined in Section 2.1.3 as $V_\Gamma^m(B)$ and $V_\Gamma^l(B)$, respectively.

[60] Or a few of them.

[61] From condition `(pi).loan[B0][tau] >= v`

[62] From condition `(gamma').C(B0) <= CMin`

prevent the whole seizable collateral amount to be atomically liquidated, by setting *Maxliq* which is respectively variable in Compound [42] and constant (equals to 0.5) in Aave [13]. In order to faithfully reproduce the real platforms behaviour, this model employs the parameter *Maxliq*, discussed in Section 2.1.3.

4.1.2 The proposed liquidating heuristic

As it is evident from Figure 4.1, the borrower's collateralization is re-established for agent A, whereas liquidations cause B and C to lose their entire collateral, disincentivizing them from repaying the loans. In light of this fact, it is sensible to pose as a research question whether there exist an optimal pair $(C_{\min}, r_{\text{liq}})$ s.t. the number of non-recoverable loans is minimal. In order to answer this question, the current section proposes a heuristic attempting to reproduce a rational behaviour for liquidators. The employed heuristic simulate a *rational* behaviour where liquidators repay the entire borrowers' collateral. This behaviour can be defined as rational for the two contradictory reasons given below.

1. Fast liquidations have the undoubted advantage of restoring liquidity, although this generates non-recoverable loans (as for agents B and C in Figure 4.1).

2. However, fast liquidations are non-desirable whenever the borrowers have collateralization slightly below r_{liq}. In fact, in these cases a minimal price fluctuation could raise their collateralization to r_{liq} allowing the liquidators to effectively restore the agents' collateralization to $C_{\min}$.

More formally, the algorithm utilised to implement the liquidator behaviour selects the liquidate input parameters, so to maximise the value of seized collateral. Specifically, given a liquidator L, the algorithm computes the remaining four liquidate parameters: the borrower's agent identifier (B_r), the amount of loan to be repaid (v_r), the type of the asset to be repaid $(\hat{\tau}_r)$ and the one of the asset to be seized (τ'_r). The algorithm operates by first computing all valid combinations of parameters and storing them in a list. Subsequently, three procedures are applied; these modify the list and lastly extract the final solution. Algorithm 1 details the heuristic, explained in the rest of this section.

As the algorithm illustrates, for each undercollaterized borrower B_0, valid liquidate parameters are chosen as following.

Line 3 - The loan asset types $\hat{\tau}$ are computed as the intersection of the liquidator's and B_0's asset types.

Line 4 - The amount of seizable collateral and its asset type is calculated as all the pairs (v', τ') such that B_0 owns v' units of a minted token τ'.

Line 5 - The current liquidate parameters, $((B_0, v'), (\hat{\tau}, \tau'))$, is added to a list l, which is subsequently modified by two function calls.

Algorithm 1 Liquidator heuristic algorithm

1: **procedure** LIQUIDATOR(L)
2: $l \leftarrow$ emptyList()
3: `forall` $\hat{\tau} \in dom(\sigma(L)) \cap \bigcup_{B_0 \,|\, C_\Gamma(B_0) < C_{\min}} dom(\pi_l(B_0))$:
4: `forall` $(v', \tau') \in \bigcup_{B_0 \,|\, C_\Gamma(B_0) < C_{\min}} \{\text{filterMinted}((cod(\sigma(B_0)), dom(\sigma(B_0))))\}$:
5: $l \leftarrow l \,^\frown\, ((B_0, v'), (\hat{\tau}, \tau'))$ ▷ Append combination to list
6: $l \leftarrow$ `updateRepaidUpToCMin`(l)
7: $l \leftarrow$ `filterRepayable`(l)
8: $elem \leftarrow$ `getMaxMinted`(l)
9: $((B_r, v_r), (\hat{\tau}_r, \tau'_r)) \leftarrow$ `seized2Repaid`$(elem)$
10: **return** $((B_r, v_r), (\hat{\tau}_r, \tau'_r))$

Line 6 - The `updateRepaidUpToCMin` function modifies an element in the list, $((B_0, v'), (\hat{\tau}, \tau'))$, if and only if $C_\Gamma(B_0) \in [r_{\text{liq}}, C_{\min})$. Graphically, this situation is illustrated by $C_\Gamma(A)$ in Figure 4.1. Accordingly to that figure, `updateRepaidUpToCMin` replaces v' with a value such that B_0's collateralization in the next configuration, Γ', is $C_{\Gamma'}(B_0) \leq 1.5$.

Line 7 - The `filterRepayable` operator removes from l the elements $((B_0, v'), (\hat{\tau}, \tau'))$ s.t. $v' : \tau$ is higher than the amount L is capable to repay.

Line 8 - The `getMaxMinted` function finds the element $elem = ((B_0, v'_m), (\hat{\tau}_r, \tau'_m))$, s.t. the value of $v' : \tau'$ is the maximum. In other words, this function call maximises the amount of seized collateral.

Line 9 - The `seized2Repaid` function converts the amount of seizable collateral $((v'_m : \tau'_m))$ into the amount of repayable loan $((v_r : \tau_r))$.

It should be noted that this algorithm selects the entire amount of the borrower's repayable loan in a given asset type. In order to limit this amount by the *Maxliq* factor, the way the algorithm has been implemented is slightly more complex. This is explained in Section 4.1.3.

In conclusion, plausibly assuming that the amount of assets types in an agent wallet is constant, it can be observed that the cost of Algorithm 1 is linear in the number of undercollaterized borrowers. This comes as a simple consequence of the fact that firstly Lines 3 and 4 could easily be translated into two separate looping constructs, secondly the function calls in Lines 6 to 8 are linear in the length of l.

4.1.3 Heuristic implementation

The LP specifications implement Algorithm 1 as shown in Listing 4.1.

```
1  eq (gamma).selectLIQParams(L, aSet, maxRep%) =
2     ((gamma).lp).seized2Repaid(
3       findMax4TV^m(
4         (gamma).filterRepayable(maxRep%, L,
5           (gamma).updateRepaidUpToCMin(
6             combine(
7               (gamma).getAFLTPairs( aSet, L ),
8               (gamma).getAMWTPairs( aSet )
9             ) ) )
10          ), (gamma).Rliq ) .
```

Listing 4.1: Maude definition of the logic for liquidate parameters selection, extract from Listing A.8, Line 328

As mentioned in the previous section, the possible combinations of parameters are generated by splitting their generation routine in two different function calls (`getAFLTPairs` and `getAMWTPairs`), whose output is then aggregated by `combine`. Apart from `maxRep%`, the remaining part of the listing resembles the algorithm, hence

it is not discussed. The input parameter `maxRep%` is a float number modelling L's liquidating speed, ergo the maximum percentage of seizable collateral. As per the current development, this value cannot exceed `Maxliq`[63], set by default to 0.5[64].

The algorithm implementation was simplified by the development of higher-order functions in Maude, inspired by [22]. A wide range of operators were defined by making use of the expressive power of higher order functions, including `map` and `filter`. Listing 4.2 shows the implementation of `filter`.

```
1  fmod AP{X :: TRIV, Y :: TRIV} is
2  sort Func{X, Y} .
3  op _[_] : Func{X, Y} X$Elt -> Y$Elt [prec 17] .
4  endfm
5
6  fmod HO1{X :: TRIV} is
7  inc LIST{X} .
8  endfm
9
10 fmod HO-FILTER{X :: TRIV} is
11 including HO1{X} .
12 protecting BOOL .
13 inc AP{X, Bool} .
14
15 var E : X$Elt .
16 var L : List{X} .
17 var P : Func{X, Bool} . --- predicate
18
19 op filter : List{X} Func{X, Bool} -> List{X} .
20 eq filter(nil, P) = nil .
21 ceq filter(E | L, P) = E | filter(L, P)
22     if P[E] . --- if predicate holds for E then add E to L
23 --- otherwise E is not added
24 eq filter(E | L, P) = filter(L, P) [owise] .
25 endfm
```

Listing 4.2: Maude definition of the `filter` higher order function, extract from Listing A.2, Line 335

As the listing shows, `filter` takes two input parameters: a term `L` of sort `List{X}` and a term `P` of sort `Func{X, Bool}`, also called a predicate. The operator returns a list containing the `L`'s elements which are satisfying `P`. Listing 4.3 illustrates the `filter` usage, by displaying the implementation of `filterRepayable`, whose behaviour was described in the previous section.

[63] This is shown in Listing A.7, Line 162.
[64] This is shown in Listing A.7, Line 141.

```
1       eq (gamma).filterRepayable(maxRep%, LOR, aFFMList) =
2          filter (aFFMList,
3                      (gamma).isLoanRepayable(maxRep%, LOR)) .
```

Listing 4.3: Example of the Maude `filter` higher order function, extract from Listing A.8, Line 194

The `filterRepayable` operator utilises the predicate `isLoanRepayable`[65] for filtering the `aFFMList` list composed by the liquidate parameters candidates.

4.2 Prices modelling

This section describes the price model employed to predict cryptocurrencies prices, based on historical data. Section 4.2.1 overviews the price model and motivates its adoption. Afterwards, Section 4.2.2 presents the three model instantiation scenarios used in the subsequent statistical analysis.

4.2.1 Predicting cryptocurrency prices

The cryptoassets prices are derived from a statistical model representative of the past price behaviour: the Geometric Brownian Motion (GBM) introduced in Section 2.4.1. A GBM is instantiated by two parameters drift and volatility which can be estimated from the currency historical data. This makes the Geometric Brownian Motion the ideal stochastic process for modelling stock prices based on their past evolution [21].

Aiming at stress-testing the LP protocol and inspired by [26], three different scenarios are designed, each comprising a pair of price trends. In practice, each scenario simulates the evolution of prices of a given collateral and loan assets, in a way that respectively when the former declines, the latter increases. In fact assuming that each borrower B_0 owes a loan in only one asset type τ_l[66] and similarly holds collateral of only one asset type τ_m[67], such a model for prices necessarily causes some borrowers to become undercollaterized, as shown in (4.1).

$$C_\Gamma(B_0) = \frac{V_\Gamma^m(B_0)}{V_\Gamma^l(B_0)} \xrightarrow{p(\tau_m)\to 0 \;\; p(\tau_l)\to V} 0, \text{with } V \gg 0 \tag{4.1}$$

More precisely, prices modelling is achieved by opportunely gathering the data used to estimate the parameters (drift and volatility) for generating a growing, decreasing or relatively constant Geometric Brownian Motion process. In the literature, daily closing prices of stock markets are utilised, given the fact that their samples generally tend to be normal, which allows to employ the GBM generic formula in Section 2.4.1. Ultimately, since prices' predictions pairs should variate in a way that they simultaneously display an opposite behaviour, it is necessary to correlate them, as shown in [31].

[65] Defined in Listing A.8, Line 179.
[66] Also called a loan asset.
[67] Also called a collateral asset.

4.2.2 Prices model instantiation

Given a collateral asset τ_m and a loan asset τ_l, the three prices evolution pairs are shown in Table 4.1.

Scenario	τ_m	τ_l	$p(\tau_m)$	$p(\tau_l)$
eth-wbtc	ETH	WBTC	Declining	Increasing
eth-usdc	ETH	USDC	Declining	Constant
usdc-wbtc	USDC	WBTC	Constant	Increasing

Table 4.1: The three implemented prices evolution scenarios

The choice of the cryptocurrencies in the table[68] is motivated by their closing price historical evolution in three different trimesters, shown in Figure 4.2. By using those samples, it is possible to simulate the desired trends indicated in the columns named $p(\tau_m)$ and $p(\tau_l)$. This is achieved by estimating the expected price returns (μ) and the price volatility (σ), which are utilised as the drift and volatility instantiating the resulting GBM. The two parameters are estimated according to [31]. The drift μ is simply obtained by computing the mean over the closing prices[69]. Contrarily, σ is calculated accordingly to (4.2)[70], where s indicates the standard deviation of the log returns and $\sqrt{T}$ is the annualisation constant.

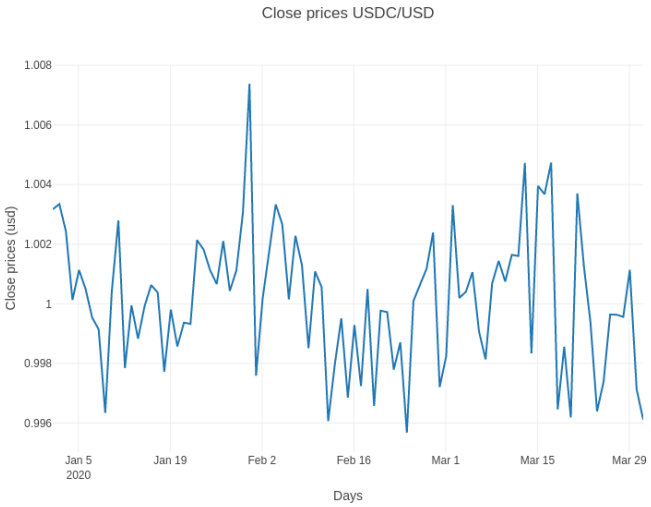
$$\sigma = \frac{s}{\sqrt{T}}, \text{ with } T = \frac{91}{365} \tag{4.2}$$

It should be noted that the selected sampling time span (91 days, i.e. a trimester) is motivated by the fact that cryptoassets are subject to sudden fluctuations and, even though short samples might not be representative of the entire population, this is a consolidated practice [31]. Besides, the resulting price predictions span over the same time frames, as each price model instantiation produces 91 prices predictions, as illustrated in Section 4.3.4. Noteworthily, the selected cryptocurrencies (ETH, USDC and WBTC) were among the four-most-utilised assets on the Compound market [20] at the moment of writing. Lastly, the selected closing price samples are suitable, since the derived log returns distributions tend to be normal (Figure B1).

Table 4.2 shows an estimation of the GBM parameters obtained from the close prices in Figure 4.2, by the previously discussed methodology. The parameters are then utilised to instantiate the six GBM processes (each for price evolution), simulating the scenarios in Table 4.1. Finally, the asset initial price P_0 is a constant set to the actual price in USD of each asset on May 5th, 2021[71].

[68] Ethereum (ETH), USD Coins (USDC) and Wrapped Bitcoins (WBTC).
[69] Section 15.3 of [31].
[70] Section 15.4 of [31].
[71] According to CoinGecko APIs.

(a) 13/01/2018-14/04/2018



(b) 01/01/2020-01/04/2020

(c) 24/11/2020-23/02/2021

Figure 4.2: Trimester closing prices, collected from CoinGecko APIs

Cryptocurrency	μ	σ	P_0 (usd)
ETH	-0.01207	0.1278	3269.08
USDC	-7.8411E-5	0.0056	0.999319
WBTC	0.01269	0.0947	57260.0

Table 4.2: Geometric Brownian Motion parameters used for the three price model instantiation scenarios

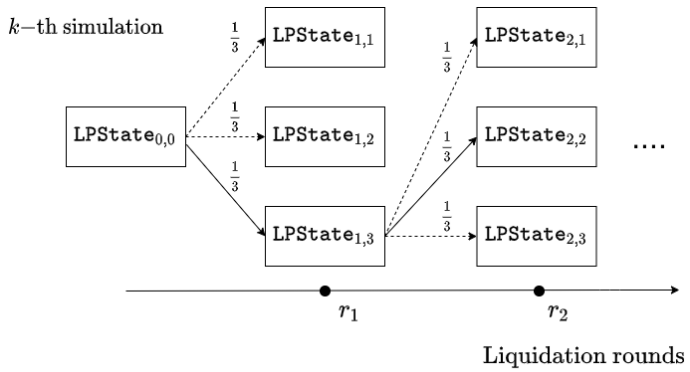


Figure 4.3: High-level view of a MultiVeStA trace, generated by the LP Maude simulator

4.3 LP model integration with MultiVeStA

This chapter discusses the interaction between the LP simulator and MultiVeStA. First, Section 4.3.1 offers a synopsis of a MultiVeStA simulation invoking the LP simulator. Second, Section 4.3.2 explains how the features discussed in Sections 4.1 and 4.2 are required by the integration. Third, Section 4.3.3 utilises the notions introduced in Section 4.3.2, in order to offer a more detailed description of the MultiVeStA simulations. Finally, Section 4.3.4 illustrates an example of MultiVeStA analysis, useful to verify the correctness of the price models instantiated in Section 4.2.2.

4.3.1 Simulating LP in MultiVeStA: overview

As introduced in Section 2.3.3, a MultiVeStA simulation can be viewed as an execution trace in the underlying transition system, having `NewState` objects as nodes. An `LPState` is a Java class extending the MultiVeStA `NewState` class. Precisely, the `LPState` can be thought as a wrapper of the model which MultiVeStA is supposed to analyse. An equivalent way of viewing `LPState` is as the interface used by MultiVeStA to interact with the model. Consequently, a `NewState` might be the result of few states and transitions of the underlying model. This is the case for the `LPState`, which this subsection only aims at giving an overview of.

In the case of the LP model, a MultiVeStA simulation resembles the graph in Figure 4.3. Here the dashed arrows show the next possible states, whereas the solid ones indicate the next-traversed state in the current simulation.

In the figure, a step of a MultiVeStA simulation leads to an insertion of a new

liquidate message in the current LP configuration. These messages are automatically generated by a specific type of agents (so called liquidators). Additionally, each simulation step provokes a change of price for both the collateral and loan assets. The implementation of both these changes to the underlying LP configuration are detailed in Section 4.3.2.

4.3.2 Liquidators and prices predictions in Maude

The operator `selectLIQParams` in Listing 4.1 is used by a liquidator for issuing a liquidate. Liquidators behaviour is implemented by the Maude rule in Listing 4.4.

```
1  --- Note: LOR -> liquidating agent identifier
2  ---       LED -> liquidated agent identifier
3  crl [issue-liquidate] :
4      [ gamma
5      < LOR : LIQ-Speed(s) | * sigma > ]
6      --- filterMsgOut removes all previous liquidate actions
7      --- issued by LOR over LED
8      => [ filterMsgOut( gamma, (LIQ, (LOR, LED)) )
9         < LOR : LIQ-Speed(s) | * sigma >
10        liquidate(LOR, LED, (v, hTau), tau')
11        ]
12     if ([gamma]).isRewritten /\ --- this ensures that 2
13                                 --- liquidates can't be
14                                 --- issued
15        LED-candidates :=
16        ( getUndercoll(gamma) \ getCollNegligible(gamma)) /\
17        ((LED, vTmp), (hTau : tau')) :=
18        ( gamma
19          < LOR : LIQ-Speed(s) | * sigma >
20            ).selectLIQParams(LOR, LED-candidates, s) /\
21        ((LED, vTmp), (hTau : tau')) =/= dummyMax4Tuple /\
22        LOR =/= LED /\ --- agent can't liquidate itself
23
24        --- Note: v:=... allows to liquidate everything
25        ---       in case vTmp * its price is below a
26        ---       threshold, owise computation would not
27        ---       converge
28        v := (((vTmp * ((gamma).lp).price[hTau]) < s) ?
29        vTmp : (vTmp * s) ) .
```

Listing 4.4: Maude definition of the logic for issuing a liquidate, modified extract from Listing A.8, Line 395. The only modification to the appendix is the renaming of `tau` in `hTau`. This is to align the notation with Figure 2.1.

Notably, an agent, having `CId` equals `LIQ-Speed(s)`, where `s` is a term of sort `Float0+`, is an agent regarded to as a liquidator. Given a liquidator, the `issue-liquidate` rule has the only effect of inserting a new liquidate to the current configuration. The insertion only occurs if the inserted liquidate was the only one in the configuration. This is guaranteed by `[gamma].isRewritten`, which ensures that

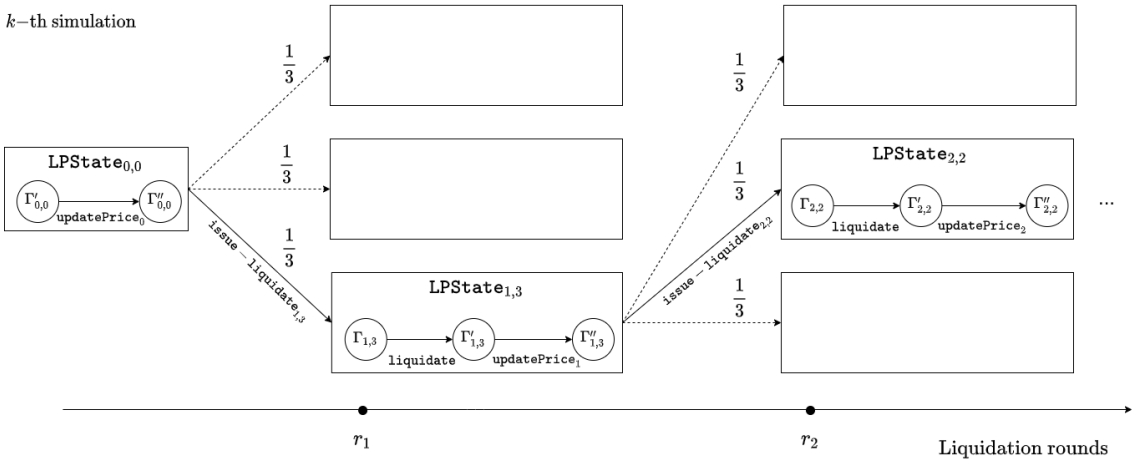


Figure 4.4: Fully-detailed representation of the MultiVeStA simulation in Figure 4.3. This is an example of simulation of an LP model having 3 liquidators.

no other message belongs to the configuration. If that is the case, the rule computes the next liquidate input parameters by invoking the `selectLIQParams` operator[72] and it issues a liquidate message.

Secondly, the prices predictions computed via the Geometric Brownian Motion introduced in Section 4.2 are integrated in the LP configuration by the operator `updatePrice`. As Listing 4.5 illustrates, the operator simply updates the old prices with the new ones, computed by a procedure external to the Maude model.

```
1  --- MultiVeStA - new price injection
2  op updatePrice(_,_) : Map{Token, Float0+}
3                        closedConfiguration
4                        -> closedConfiguration .
5  eq updatePrice(newP, [(pi | oldP) gamma]) =
6                        [(pi | newP) gamma] .
```

Listing 4.5: Maude definition of the logic for updating the LP assets prices, extract from Listing A.7, Line 1777

4.3.3 LPState: the LP-MultiVeStA interface

The Maude functionalities discussed in the previous section, `issue-liquidate` and `updatePrice`, are directly utilised by MultiVeStA simulations, as visualised in Figure 4.4.

The graph illustrates that at each step of a simulation, liquidators initiate a race condition to have their `liquidate` message added to the next `LPState`. As result of this race condition, where all liquidators try executing the `issue-liquidate` rule,

[72] The subsequent condition where `v` is assigned to `vTmp * s`, ensures that the liquidator repays a loan based on its liquidation speed. This does not occur, only if the value of the repayable amount is almost negligible. In that case, the liquidator is allowed to repay the whole repayable amount.

only one among them manages to publish the `liquidate`. It should be noted that the `liquidate` is selected according to a uniform distribution, as shown in the figure. Subsequently, the next `LPState` consists of the previous state with the newly added `liquidate`. At this stage, first the `liquidate` is rewritten, as indicated by the edge labelled `liquidate` and second the newly sampled prices predictions for the loan and collateral assets are injected in the LP configuration. The simulation continues following this scheme until liquidators can issue `liquidate` messages, i.e. until the set of undercollaterized agents in the LP configuration becomes empty.

It is worth noting that at each step, only one liquidator can issue a `liquidate`. This is guaranteed by the first condition of the rule `issue-liquidate` in Listing 4.4. Lastly, it is now straightforward to observe that Figure 4.4 is a simulation having three competing liquidators.

4.3.4 Expected prices predictions

As mentioned in Section 2.3.3, MultiVeStA-executable analyses are defined by properties expressed in MultiQuaTEx. MultiVeStA can be employed to observe the behaviour of each component of an `LPState`. For instance, this subsection uses MultiVeStA to examine the prices predictions generated by the geometric Brownian motion in each of the scenarios explained in Section 4.2.2. Listing 4.6 is the MultiQuaTEx property querying the LP model for the mean of the step-wise asset collateral value, over 91 consecutive observations.

```
1  obsAtStep(i,x) =
2  if ( s.rval("steps") == x )
3    then s.rval(i)
4    else # obsAtStep(i,x)
5   fi ;
6  eval parametric(E[ obsAtStep("CUR_COLLATERAL_PRICE",x) ],
7                  x,1,1,91) ;
```

Listing 4.6: `meanCollateralPrice.multiquatex` - MultiQuaTEx property producing the mean collateral asset price evolution.

The main idea of this property is that at each step of a simulation, MultiVeStA questions the current `LPState` about the new value of the collateral asset price, sampled by a geometric Brownian motion. Subsequently, for each liquidation round the mean of the step-wise observations is computed and returned. This is the semantics expressed by the `E[`*obsAtStep(...)*`]` input argument to `eval parametric`. The remaining arguments to `eval parametric` repeat the same procedure (`E[`*obsAtStep(...)*`]`) for 91 liquidation rounds[73]. A similar property has also been developed for querying the mean loan asset price evolution, for 91 liquidation rounds.

Figure 4.5 shows the normalised trend of the price scenarios, discussed in Section 4.2.2. The figures show that the expected behaviour, expressed in Table 4.1 is

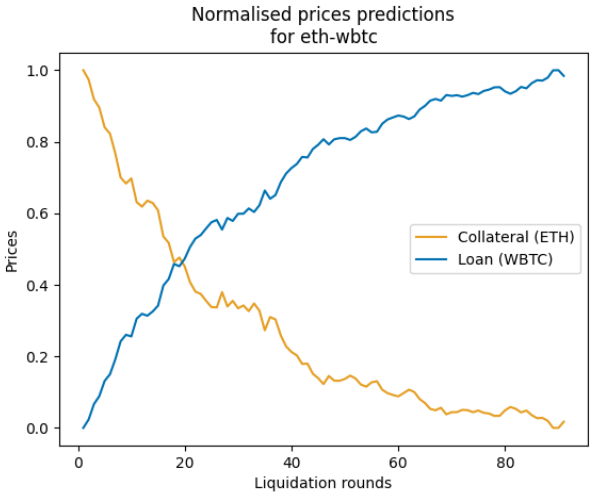
[73]The number 91 is motivated by the historical time spans considered to instantiate the geometric Brownian motion processes, discussed in Section 4.2.2.

obtained in all the considered scenarios. Additionally, in Figures 4.5a and 4.5c prices predictions are strongly correlated as it is expected. In fact, the GBMs pairs were instantiated as negatively correlated processes[74] accordingly to [31], Section 14.5. Contrarily, Figure 4.5b shows less correlated prices predictions. This is probably due to the fact that the computation was bounded to execute maximum 5010 simulations. In fact, from experimental evidence, the approximation seem to converge at a very slow speed[75].

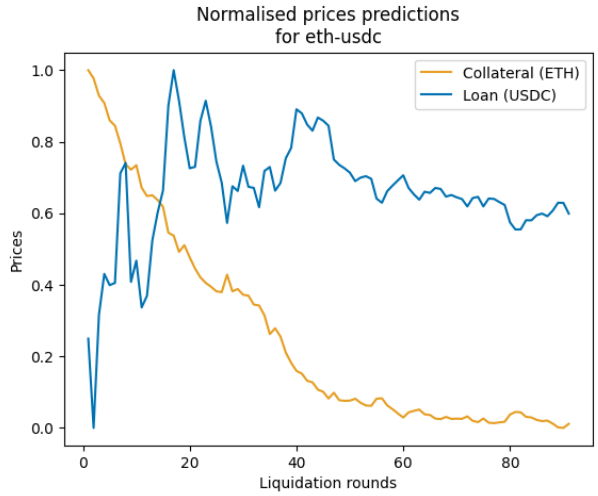
Finally, Figures B2 and B3 illustrate the unnormalised behaviour of the considered GBM prediction scenarios, w.r.t. the collateral and loan assets respectively.

[74] With $\rho = 1$, where ρ is the Pearson correlation coefficient.

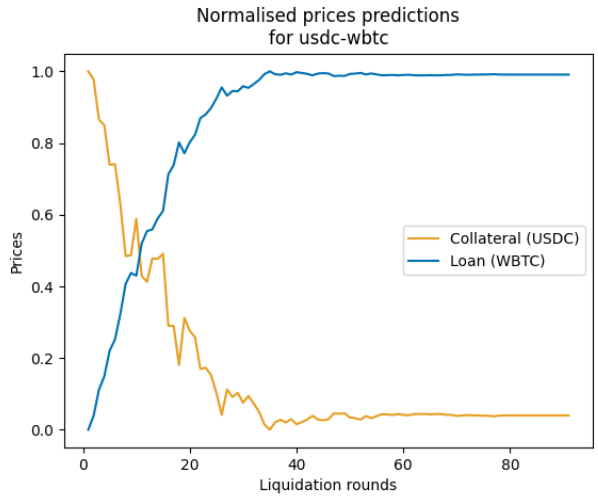
[75] Each observation would take around 2 hours to reach the respective CI.



(a)



(b)



(c)

Figure 4.5: Prices predictions produced, for each scenario in Table 4.1, by GBMs instantiated with the parameters in Table 4.2.

5 The formal verification tool

This chapter shows the functionalities of the developed verification tool for the lending pools model, as described in Sections 3 and 4.

Specifically, Section 5.1 discusses the most basic usage of the tool i.e. simulation of LP model executions. Subsequently, Section 5.2 introduces more advanced analyses such as invariant properties verification via reachability analysis and LTL model checking. Finally, Section 5.3 explains an example of MultiVeStA statistical analysis conducted on the LP simulator, developed in Section 4.3.

5.1 LP model simulation

The specifications introduced in Section 3 allow to simulate the lending pools model with a high level of fidelity and in an automated manner. This section explains possible applications offered by the Maude specification.

First, Section 5.1.1 compares an LP execution given by [7] with one produced by the Maude specification of LP. This is a basic but essential functionality of the tool, confirming or denying the correctness of an LP execution or allowing the verifier to explore its complete state space.

Second, Section 5.1.2 illustrates an example of a price oracle attack, as presented in [7]. In this case, the Maude specification represents a direct and simple means to verify the plausibility of an attack.

5.1.1 Executable specifications

The Maude specification can emulate the behaviour of the LP model in Section 2.1.3. For instance, Listing 5.1 defines `Gamma^i0`, representing the initial configuration Γ_0^i of the running example in Figure 2.2 and Table 2.3. In fact, each state resulting from a `rewrite` on term `Gamma^i0` resembles a row in Figure 2.2.

```
op Gamma^i0 : -> closedConfiguration .
eq Gamma^i0 = [({
                  fund: tau(0) |-> 3.0e+2 ;
                        tau(1) |-> 1.95e+2,
                  loan: A |-> tau(1) |-> 8.0e+1 ;
                        B |-> tau(1) |-> 1.0e+2 ;
                        C |-> tau(1) |-> 1.25e+2,
                  mint: tau(0) |-> (tau(0) ',3.0e+2) ;
                        tau(1) |-> (tau(1) ',5.0e+2)
                } |
                tau(0) |-> 1.0 ;
                tau(1) |-> 1.0 )
            < A : noState | * (tau(0) |-> 0.0 ;
                               tau(1) |-> 8.0e+1 ;
                                 tau(0)' |-> 1.0e+2) >
            < B : noState | * (tau(0) |-> 0.0 ;
                                 tau(1) |-> 1.0e+2 ;
                               tau(0) ' |-> 1.0e+2) >
```

```
            < C : noState | * (tau(0) |-> 0.0 ;
                               tau(1) |-> 1.25e+2 ;
                               tau(0)' |-> 1.0e+2) >
            < D : noState | * (tau(0) |-> 1.0e+2 ;
                               tau(1) |-> 5.0e+2 ;
                               tau(1)' |-> 5.0e+2) >
            < C(0) : Coll | * (A,1.25),
                            * (B,1.0),
                            * (C,8.0000000000000004e-1),
                            * (D,-) >
            < R(0) : Round | none >
            < P(0) : LiqParams | CMin(1.5),Rliq(1.1) >
        liquidate(D, A, (50.0, tau(1)), tau(0)')
        liquidate(D, B, (90.90909090909090, tau(1)), tau(0)')
        liquidate(D, C, (90.90909090909090, tau(1)), tau(0)')
    ] .
```

Listing 5.1: Initial configuration Γ_0^i in Figure 2.2 expressed in Maude.

To exemplify the usage of the Maude simulator, the state space of Γ_0^i is now explored via the `search` command. The execution of `search` on `Gamma^i0` resulting from zero or more rewriting steps (`=>*`) generates the terms in Listing 5.2. The `states` 0 and 7 represent the configurations Γ_0^i and $\Gamma_{3,1}$ in Figure 2.2, respectively. The full Maude representation of the running example (Figure 2.2 and table 2.3) is given in Listing C.1.

```
search in SEARCH-EXAMPLE :
       Gamma^i0 =>* X:closedConfiguration .

Solution 1 (state 0) -- Γ_0^i, in Table 2.3
states: 1  rewrites: 147 in 0ms cpu (0ms real)
(~ rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ; tau(1) |-> 1.95e+2,
          loan: A |-> tau(1) |-> 8.0e+1 ;
                B |-> tau(1) |-> 1.0e+2 ;
                C |-> tau(1) |-> 1.25e+2,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 1.0e+2) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 1.0e+2) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
```

```
                    tau(0)' |-> 1.0e+2) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 5.0e+2 ;
                    tau(1)' |-> 5.0e+2) >
< C(0) : Coll | * (A,1.25),
                * (B,1.0),
                * (C,0.8),
                * (D,-) >
< R(0) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, A, (5.0e+1,tau(1)), tau(0)')
liquidate(D, B, (9.0909090909090907e+1,tau(1)), tau(0)')
liquidate(D, C, (9.0909090909090907e+1,tau(1)), tau(0)')]
    ⋮              ⋮              ⋮

Solution 8 (state 7) -- Γ3,1, in Table 2.3
states: 8  rewrites: 9701 in 9ms cpu (10ms real)
(974093 rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 4.2681818181818176e+2,
          loan: A |-> tau(1) |-> 3.0e+1 ;
                B |-> tau(1) |-> 9.0909090909090935 ;
                C |-> tau(1) |->
    3.4090909090909093e+1,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 4.4999999999999993e+1) >
< B : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 0.0) >
< C : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.25e+2 ;
                  tau(0)' |-> 0.0) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 2.6818181818181824e+2 ;
                  tau(0)' |-> 2.55e+2 ;
                  tau(1)' |-> 5.0e+2) >
< C(3) : Coll | * (A,1.5),
                * (B,0.0),
                * (C,0.0),
                * (D,-) >
< R(3) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1) >]
```

```
No more solutions.
states: 8  rewrites: 10827 in 9ms cpu (11ms real)
(1087157 rewrites/second)
```

Listing 5.2: Results produced by a `search` simulating the traces in the transition system of Figure 2.2. Extract from Listing C.1

5.1.2 Price oracle attack

The specifications developed in Section 3 can be employed to reproduce or possibly investigate on (novel) attacks compromising the LP safety. A simple Maude `rewrite` command, explained in Section 2.2.2, is sufficient to observe the attacks effect.

Listing 5.3 describes the initial configuration for simulating the price oracle attack proposed in Section 5.1 of [7]. The initial attack assumption is that the attacker, the agent `Att`, is empowered to change token prices. As a consequence, `Att` causes a sudden decline in the price of `tau(0)` tokens, used by the victim `Vic` as its collateral. The sudden price drop entails `Vic`'s collateralization to fall below `Rliq`[76], allowing `Att` to liquidate[77] `Vic`.

Listing 5.4 depicts the results of the attack on the initial configuration. The resulting configuration is obtained by simply executing a `rewrite [2]` on the configuration `CSAttack2A` in Listing 5.3, which consumes the two `Messages` `price` and `liquidate`, in this order. As it is illustrated in the listing, `Att` is capable of seizing the entire `Vic`'s collateral (`1500` units of `tau(0)`), by not even repaying a unit of `tau(1)` B's debt.

```
op CSAttack2A : -> closedConfiguration .
eq CSAttack2A =
[ (
   {
     fund: tau(0) |-> 0.0 ;
           tau(1) |-> 1.5e+3,
     loan: Vic |-> tau(0) |-> 1.0e+3,
     mint: tau(0) |-> (tau(0) ',1.0e+3) ;
           tau(1) |-> (tau(1) ',1.5e+3)
   }
    |
   tau(0) |-> 1.0 ; tau(1) |-> 1.0
  )
   < Att : noState | * (tau(0)  |-> 1.0 ;
                        tau(0)' |-> 1.0e+3) >
   < Vic : noState | * (tau(0)  |-> 1.0e+3 ;
                        tau(1)  |-> 0.0 ;
                        tau(1)' |-> 1.5e+3) >
   < R(4) : Round | none >
```

[76] This is a consequence of the definition of collateralization (2.8), also shown by $C_\Gamma(C)$ in Figure 4.1.

[77] For simplicity, in Listing 5.3 `Maxliq` is assumed to be equal to 1.0, permitting `Att` to seize the entire `Vic`'s collateral. This is not the case in the LP implementations as described in Sections 2.1.3 and 4.1.1.

```
    < C(4) : Coll | * (Att,-),* (Vic,1.5),* (C,-) >
    < P(0) : LiqParams | CMin(1.5), Rliq(1.1) >
    price(tau(1) |-> 1.0e-24)
    liquidate(Att, Vic, ( ((1499.0 * 1.0e-24) / 1.1),
                                tau(0) ), tau(1)')
]
```

Listing 5.3: Initial configuration for simulating a price oracle attack, according to [7], Section 5.1.

```
rewrite [2] in ATTACKS : CSAttack2A .
rewrites: 381 in 3ms cpu (0ms real) (114311 rewrites/second)
[ (
    {
      fund: tau(0) |-> 1.3627272727272726e-21 ;
            tau(1) |-> 1.5e+3,
      loan: Vic |-> tau(0) |-> 1.0e+3,
      mint: tau(0) |-> (tau(0) ',1.0e+3) ;
            tau(1) |-> (tau(1) ',1.5e+3)
    }
     |
    tau(0) |-> 1.0 ;
    --- Collateral price set to 0, by the price update
    tau(1) |-> 9.9999999999999992e-25
    )
    --- The attacker successfully liquidates all Vic's
    --- collateral by not repaying its loan
    < Att : noState | * (tau(0) |-> 1.0 ;
                         tau(0)' |-> 1.0e+3 ;
                         tau(1)' |-> 1.4990000000000002e+3) >
    < Vic : noState | * (tau(0) |-> 1.0e+3 ;
                          tau(1) |-> 0.0 ;
                          tau(1)' |-> 9.9999999999977263e-1) >
    --- Vic collateralization drops to 0, after the
    --- collateral price is set to 0
    < C(6) : Coll | * (Att,-),* (Vic,9.999999999997726e-28) >
    < R(6) : Round | none >
    < P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
]
```

Listing 5.4: Final state of a price oracle attack simulation.

5.2 Model Checking invariants

This section illustrates two verification techniques in order to check whether given invariants hold in the LP model. Section 5.2.1 presents an example of reachability analysis for verifying that an invariant holds in a given initial state. Similarly, Section 5.2.2 illustrates the verification of the same invariant property expressed via a simple LTL formula.

5.2.1 Reachability analysis

The rich Maude environment offers support for proving safety properties of a model specified in Maude. This point is illustrated by conducting a simple reachability analysis on the finite state space originated from `Gamma^i0` configuration[78], defined in Listing 5.1. Similarly to the examples given in Section 5.1, this is a very simple application which could be employed for proving more complex and less obvious properties.

The analysis discussed here has been developed by integrating the Maude `MODEL-CHECKER` module, part of the native Maude library, `model-checker.maude`. Consequently, some basic notions on the integration of this specification with that module should be given. In Maude, the essential operator utilised to express a property (or proposition) holding in a model state is `op _|=_ : State Prop -> Bool`. This operator is the main service exposed by the `MODEL-CHECKER` module. In order to use it, three actions are required:

- defining the specified model configuration as a subsort of `State`;
- defining an operator returning a term of sort `Prop`;
- defining the operator `_|=_` semantics.

Listing 5.5 depicts how these actions are taken in the LP specification. Thus, from this point, any term of sort `closedConfiguration` is regarded to as a `State` term and `lemma1` term is of sort `Prop`. As a result, the term `gamma |= lemma1`, with `gamma` a `closedConfiguration`, is a well-defined term. In fact, the equation in Line 7 allows to reduce `gamma |= lemma1` to `testLemma1(gamma)`, i.e. the operator returning true iff `gamma` satisfies `lemma1`[79].

```
1  var obj : Object .
2  var msg : Msg .
3  var bal : Map{Token, Float0+} .
4  var agState : Agent-State .
5
6  op testLemma1 : closedConfiguration -> Bool .
7  eq testLemma1(gamma) =
8              $testLemma1((gamma).config,
9                          ((gamma).config).mintedTokens) .
10
11 var mTokenSet : Set{Minted-Token} .
12 var tau' : Minted-Token .
13
14 --- pre: 1. Set{Minted-Token} is (config).mintedTokens
15 ---      2. minted token tau is tau'
16 op $testLemma1 : Configuration Set{Token} -> Bool .
17 eq $testLemma1(config, empty) = true .
18 eq $testLemma1(config, tau') =
```

[78] This configuration implies a state space comprising 7 other configurations.
[79] Defined in Listing A.9, Line 142.

```
19              (config).balSum(tau') ==
20                 snd((config).pi.m[(config).u(tau')]) .
21  eq $testLemma1(config, (mTokenSet, tau')) =
22      if ((config).balSum(tau') ==
23          snd((config).pi.m[(config).u(tau')])) then
24          $testLemma1(config, (mTokenSet))
25      else
26          false
27      fi .
       ⋮                ⋮                ⋮

29  subsort closedConfiguration < State .
30  op lemma1 : -> Prop .
31
32  --- Given gamma a variable of sort closedConfiguration
33  eq gamma |= lemma1 = testLemma1(gamma) .
```

Listing 5.5: Statements required to integrate LP specification with the `MODEL-CHECKER` native Maude module. Extract from Listing A.9, Lines 14 and 142.

Having established this, Listing 5.6 illustrates an example where the state property of Lemma 1 in [7][80] `lemma1` is shown to hold for any configuration reachable from `Gamma^i0`[81].

```
search in SEARCH-EXAMPLE : Gamma^i0 =>* C:closedConfiguration
      such that not C:closedConfiguration |= lemma1 = true .
```

Listing 5.6: Example of search used to verify a safety property.

In fact, Listing 5.7 shows that no reachable configuration satisfies `not lemma1`. This signifies that the negation of `lemma1` does not hold in any reachable state or, equivalently, `lemma1` holds in all the states reachable from `Gamma^i0`.

```
No solution.
states: 8  rewrites: 7080 in 6ms cpu (7ms real)
(1071266 rewrites/second)
```

Listing 5.7: Output returned by the `search` command in Listing 5.6.

5.2.2 LTL model checking

The Maude `MODEL-CHECKER` module also defines the Linear Temporal Logic constructs. As a result the command in Listing 5.6 can be expressed by a simpler Maude command containing an LTL formula. Additionally, the Maude LTL model checker provides more efficient procedures to check properties satisfiability [23].

[80] This lemma was proven by Bartoletti et al., hence the above example is purely illustrative.
[81] Defined in Listing 5.1.

Listing 5.8 illustrates an alternative command to check that the state formula underlying `lemma1` holds in all reachable states of `Gamma^i0`. In Line 1 of the listing, the checked LTL formula is `[] lemma1`. The property, checked in the initial configuration `Gamma^i0`, is to be interpret as `lemma1` *holds in any path originating in* `Gamma^i0`. Therefore, the `[]` operator is the Maude implementation of the $\Box$ temporal operator, as defined in Section 2.3.2.

```
1  reduce in SEARCH-EXAMPLE : modelCheck(Gamma^i0, []lemma1) .
2  rewrites: 12766 in 16ms cpu (15ms real)
3  (770428 rewrites/second)
4  result Bool: true
```

Listing 5.8: Example of LTL formula used to verify a safety property.

Finally, it should be noted that any formula expressible in LTL, not just invariants, may be checked of the model as long as the state space derived from the chosen initial configuration is finite [19], Chapter 12.

5.3 Statistical analyses

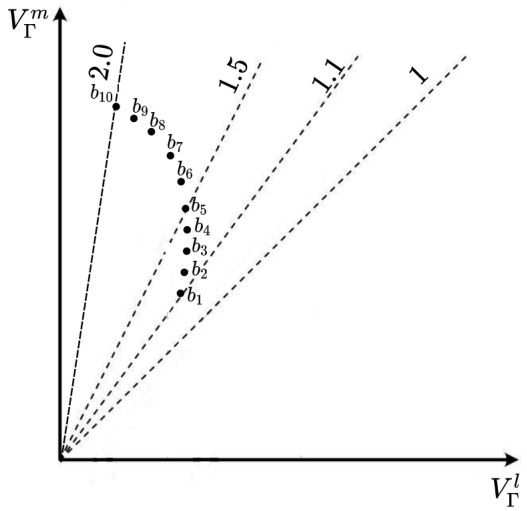


Figure 5.1: Distribution of the borrowers' collateralization in the initial configurations.

This section describes the experiments conducted on the LP model simulator, described in Section 4, in order to answer the question: *given some "undesirable" scenarios, what is the "optimal" pair of LP parameters* $C_{\min}$ *and* r_{liq}*?* However, to elaborate on the results, it is appropriate to summarise the assumptions which have been set so far. A convenient summary can be offered by a better definition for the two adjectives *"undesirable"* and *"optimal"*:

"undesirable" - in this study such scenarios are generated by four factors. First, the liquidator logic defined in Section 4.1, determines immediate and quick[82] liquidations, causing a significant financial loss to the liquidated party. Secondly, the agent to be liquidated is selected so to maximise the value of seized collateral, which is the most beneficial and rational option for liquidators. Thirdly, liquidators are assumed to hold an *infinite* amount of resources, which allows them to repeat liquidations as long as there exists an undercollaterized agent. Finally, cryptoasset prices evolve following a trend aimed at causing borrowers to suddenly become undercollaterized.

"optimal" - the pair $C_{\min}$ and r_{liq} is optimal if it minimises the number of undercollaterized borrowers. This pair is estimated by executing MultiVeStA experiments for all $C_{\min}$ ranging, with step 0.1, from 1.2 to 1.5 and r_{liq} ranging from 1.1 to $C_{\min} - 0.1$. These ranges were selected based on the values assigned to these parameters in the real implementation: $C_{\min} = 1.5$ and $r_{\text{liq}} = 1.1$ [7].

On these premises, Section 5.3.1 illustrates the LP model initial configurations used for the subsequent experimentation. Next, Section 5.3.2 will present the results of the performed experiments.

5.3.1 Initial configurations for experiments

The initial configuration were designed so to test the resistance of different borrowers' collateralization to becoming unrecoverable, when subject to repeated liquidations. Since the intention is to observe the model behaviour under three price models (Section 4.2.2), three different initial configurations are produced, each having a different price for collateral and loan assets. Nevertheless all the configurations share the same amount and types of agents. Specifically, a generic initial configuration comprises ten borrowers having collateralization ranging from 1.0 to 2.0, with step 0.1. This is depicted in Figure 5.1, where b_i represents the generic borrower B_i's collateralization ($C_{\Gamma^i}(B_i)$), for Γ^i initial configuration.

Additionally, an arbitrary number of liquidators (three) are added to each configuration. This determines a race condition at each simulation step, as shown in Figure 4.4.

5.3.2 Experimental results

```
1  obsAtStep(i,x) =
2  if ( s.rval("steps") == x )
3    then s.rval(i)
4    else # obsAtStep(i,x)
5  fi ;
6  eval parametric(E[ obsAtStep("1_COLLATERALIZATION",x) ],
7                  E[ obsAtStep("2_COLLATERALIZATION",x) ],
8                  E[ obsAtStep("3_COLLATERALIZATION",x) ],
```

[82] The repayed debt amounts to `Maxliq` $\cdot\, t$, with t total amount of repayable debt.

```
9                E[ obsAtStep("4_COLLATERALIZATION", x) ],
10               E[ obsAtStep("5_COLLATERALIZATION", x) ],
11               E[ obsAtStep("6_COLLATERALIZATION", x) ],
12               E[ obsAtStep("7_COLLATERALIZATION", x) ],
13               E[ obsAtStep("8_COLLATERALIZATION", x) ],
14               E[ obsAtStep("9_COLLATERALIZATION", x) ],
15               E[ obsAtStep("10_COLLATERALIZATION", x) ],
16               x, 1, 1, 91) ;
```

Listing 5.9: `perAgentCollateralization.multiquatex` - MultiQuaTEx property producing the mean collateralization per agent (maximum 10 borrowers).

The results discussed in this section were obtained by performing MultiVeStA experiments of the LP simulator. The evaluation was performed on an 8GB-RAM machine with two 1.20GHz cores. Specifically, the inputs to the tool are: the LP simulator discussed in Sections 3 and 4, the MultiQuaTEx property in Listing 5.9 and a pair of statistical parameters (α, δ), discussed below. Consequently, MultiVeStA potential of generating statistically-valid approximations is entirely employed, as no bound to the number of simulations has been set, instead the simulator runs until the required confidence interval (CI) is reached[83]. The next two paragraphs describe the inputs to MultiVeStA, afterwards the results are introduced and discussed.

Listing 5.9 is the MultiQuaTEx property to be estimated. The property semantics is to compute the expected collateralization value at each liquidation round and for each borrower in the initial configurations, discussed in Section 5.3.1.

Figure 5.2 shows an example of the variables approximated by a bounded MultiVeStA execution[84]. Given the nature of the analysed variables in the figure, $\delta = 0.05$ appears to be an acceptable confidence interval width. Additionally, the CI confidence level is set to $\alpha = 0.05$ so to obtain 95% statistical confidence.

Figures 5.3a to 5.3c illustrate the per-borrower collateralization in the eth-wbtc prices scenario, for varying liquidation rounds and `CMin-Rliq` choices. Observing the corresponding surface graphs in each prices scenarios (Figures B5a to B5c), it is evident that undercollaterized agents have a very different behaviour than overcollaterized ones. Specifically, the undercollaterized agents undergo very serious liquidations, which often lead them to unrecoverability, as their collateralization converges to a constant below $C_{\min}$. Contrarily, overcollaterized agents do not incur in severe financial losses.

Additionally, Figures 5.4a to 5.4c show that the `CMin-Rliq` having the least negative effects on undercollaterized balances is `CMin` $= 1.5$, `Rliq` $= 1.1$. This is also quantitatively confirmed by the figures in Table 5.1. Intuitively, this is a consequence of the fact that when $C_{\min} = 1.5$ and $r_{\text{liq}} = 1.1$ the collateralization of each agent b_1 to b_5 is higher on average than for any other $C_{\min}$ and r_{liq} pairs. As a result, the number of unrecoverable loans, the ones held by agents whose collateralization is below 1, is minimised.

[83] Figure B4 shows the amount of simulations required to compute the results for each prices scenario.

[84] With $\alpha = 0.05, \delta = 0.01$ and $n = 5010$, maximum number of executions.

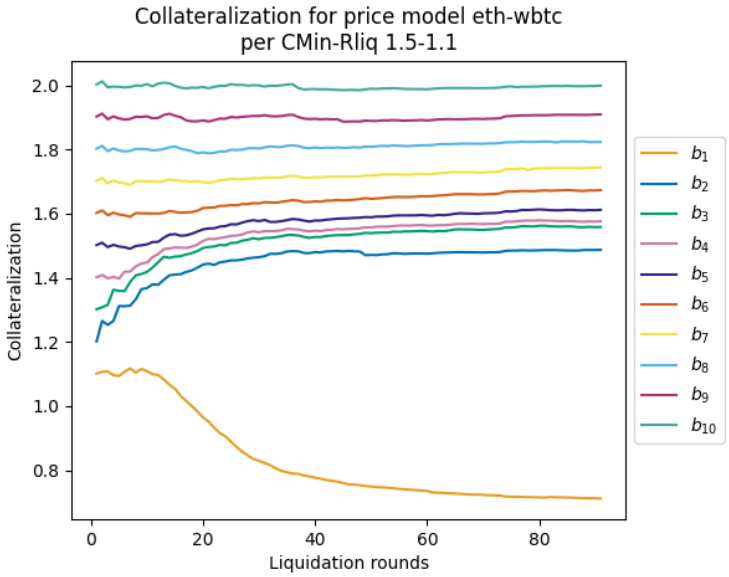


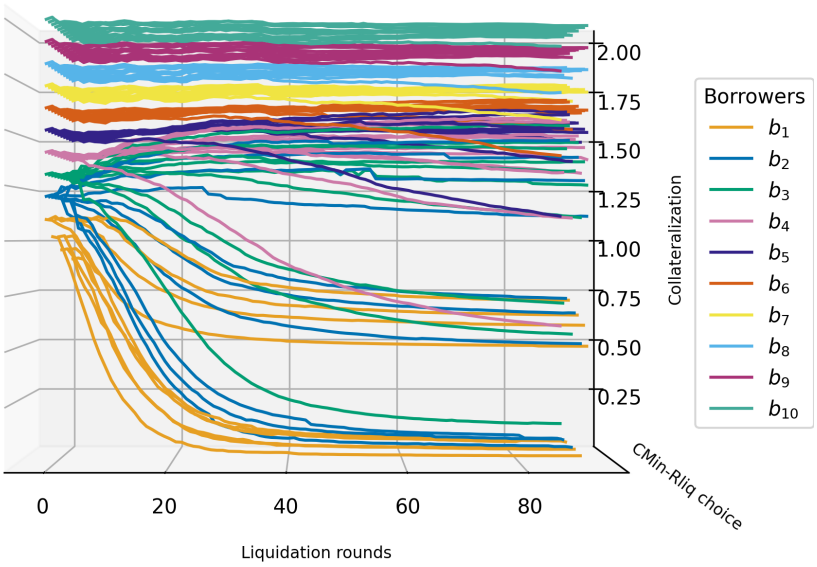
Figure 5.2: Expected collateralization per agent at each liquidation round.

Finally, Figures 5.5a to 5.5c display that overcollaterized borrowers could still incur in liquidations, in case the prices abruptly change as in the prices scenario eth-wbtc[85]. Differently, in the other scenarios, employing the stable coin usdc, overcollaterized agents are, on average, rarely liquidated.

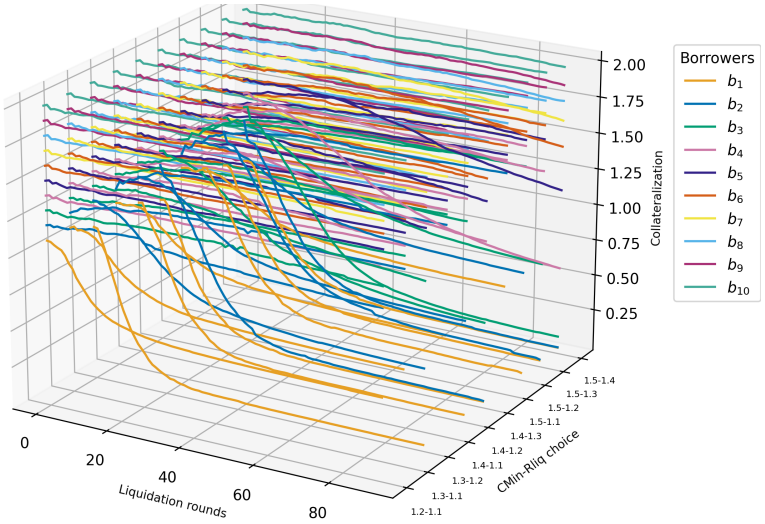
Price scenario	`(CMin-Rliq)`		
	(1.5-1.1)	(1.4-1.1)	(1.3-1.1)
eth-wbtc	0.7115	0.6518	0.6137
eth-usdc	0.7106	0.6583	0.6231
usdc-wbtc	0.8381	0.7739	0.7299

Table 5.1: Minimum average $C_\Gamma(B_1)$ per liquidation round for the three `(CMin-Rliq)` maximising $C_\Gamma(B_1)$.

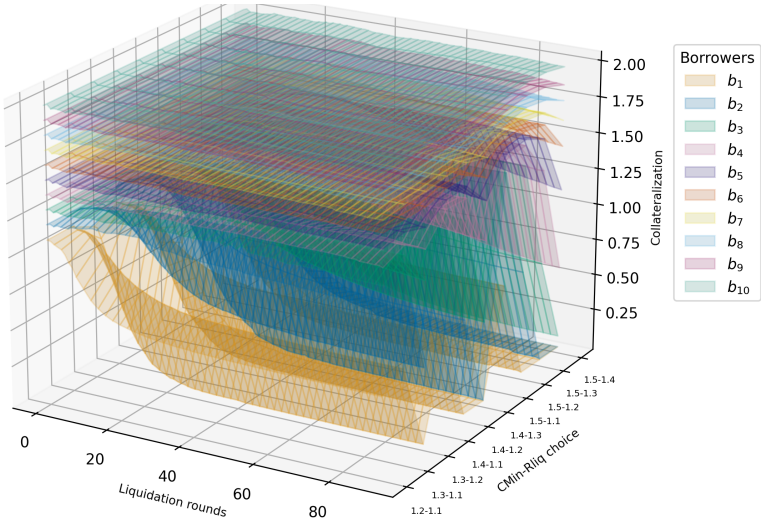
[85]This is mainly determined by the opposite drift and high volatility used to model the ETH and WBTC prices evolution (Table 4.2).



(a) Line graph for varying liquidation rounds.

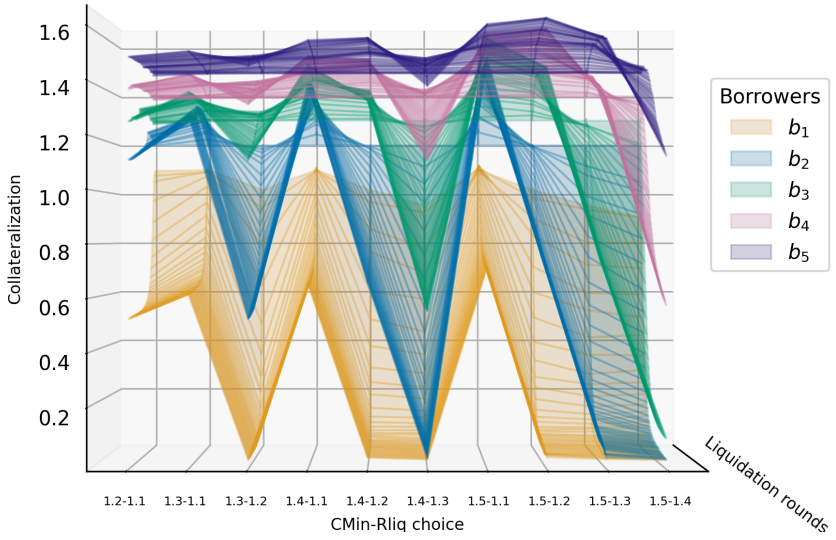


(b) Figure 5.3a, rotated by 45 degrees around the z-axis.

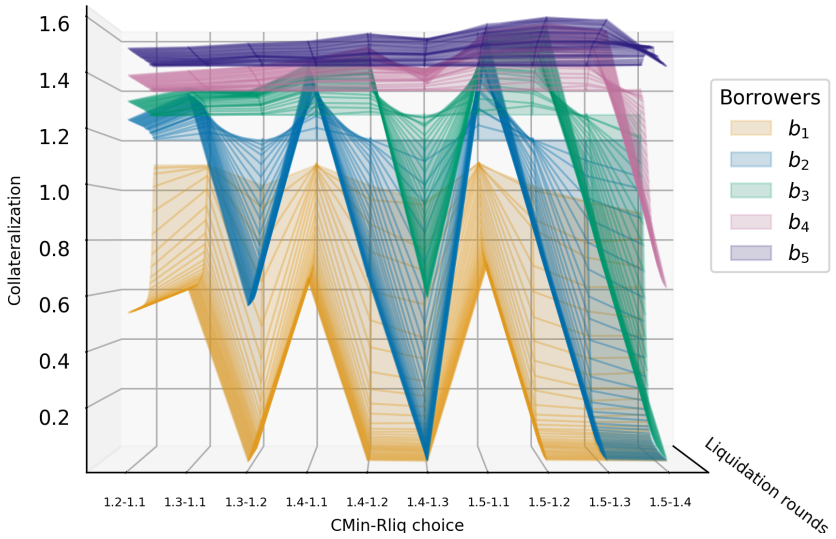


(c) Surface graph corresponding to Figure 5.3b.

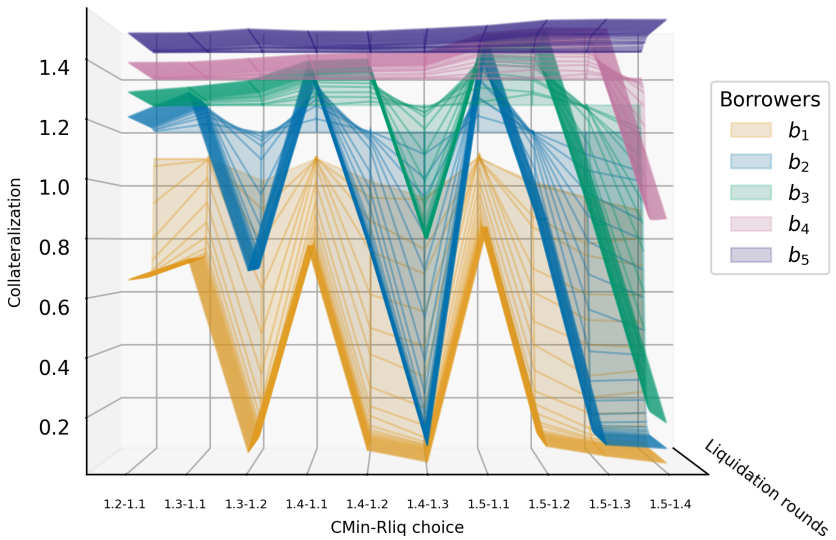
Figure 5.3: Per-borrower collateralization (b_1 to b_{10}) in the eth-wbtc prices scenario, for varying liquidation rounds and `CMin-Rliq` choices.



(a) Scenario: eth-wbtc.

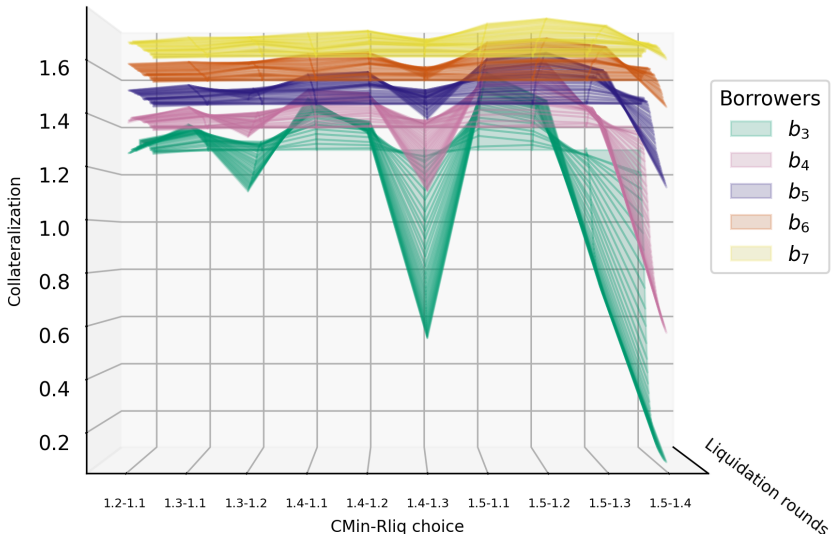


(b) Scenario: eth-usdc.

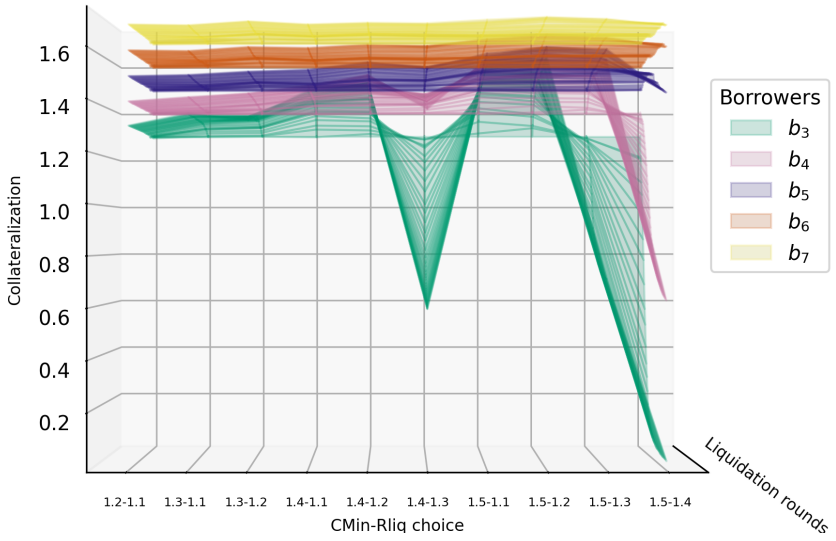


(c) Scenario: usdc-wbtc.

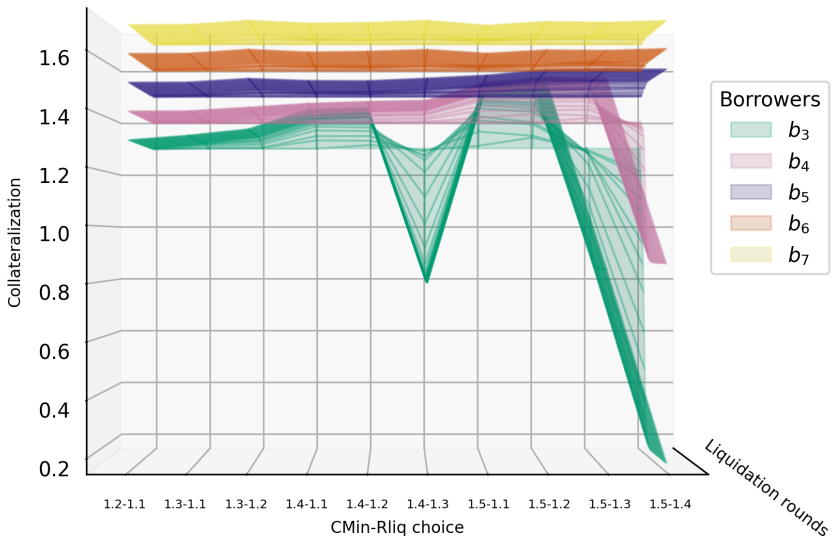
Figure 5.4: Per-borrower collateralization (b_1 to b_5) in the three prices scenarios, for varying `CMin-rliq` choices.



(a) Scenario: eth-wbtc.



(b) Scenario: eth-usdc.



(c) Scenario: usdc-wbtc.

Figure 5.5: Per-borrower collateralization (b_3 to b_7) in the three prices scenarios, for varying `CMin-rliq` choices.

6 Conclusions

This chapter offers additional context to the specific research questions addressed by the thesis. This is achieved firstly by Section 6.1, discussing and comparing other research work. Secondly, Section 6.2 explains the limitations of this work and attempts to offer solutions to these issues. Thirdly, Section 6.3 proposes ideas for future research which could be conducted with the use of the developed tool.

6.1 Related work and contributions

Verification of DeFi Smart Contracts is a fairly recent research area where several techniques have been applied, according to Liu et el. [36]. They classify the contributions into two categories based on the investigation scope: program-based and behaviour-based. The former research corpus investigates the correctness of specific smart contracts programs, whereas the latter develops formal models for them and considers significant properties of these models. Given the focus of this thesis on developing a verification tool for an LP behavioural model, the rest of this section considers only behaviour-based pieces of research.

Behaviour-based formal verification primarily follows two parallel directions:

1. verification of the model properties [11, 5, 2, 52];
2. statistical analysis of the model variables [4, 17, 33, 16, 26].

To the best of the author's knowledge, Bigi et al. [11] presented the first work on smart contract properties formal verification. Their study combines a game-theoretic approach with probabilistic model checking, ultimately validating their results with the model checker PRISM [34]. Using a similar methodology, Bai et al. [5] investigated properties of these systems using the model checker SPIN [29] in order to develop secure templates for smart contracts. Another example of research in this direction is Tolmach et al. [52] which developed the first multi-pools model and verified invariant properties initially formulated by [9]. Finally, Abdellatif and Brousmiche [2] proposed a very relevant study on smart contracts, by modelling not only the contracts and the agents' behaviour but also the underlying blockchain by the Behaviour Interaction Priorities (BIP) framework [8]. This work is also related to the thesis as it made use of *Statistical Model Checking* (SMC) [30], a formal verification technique which has strongly influenced the development of MultiVeStA [48, 53]. In fact, SMC verifies probabilistic properties of a model by using statistical means, i.e. Monte-Carlo simulations, whose number is estimated on-the-fly. The estimation is based on two input parameters the risk level, α, and the precision, δ. Then, if p and p' are, respectively, the real and the estimated probabilities for the property ϕ to hold in a given state, simulations are performed until $\mathbb{P}(|p - p'| \leq \delta) \geq 1 - \alpha$ holds [2]. Thus, this methodology clearly resembles the technique employed by MultiVeStA, shown in Section 2.3.3.

The more recent direction of studies on smart contracts is also loosely relevant to Abdellatif et al. [2], as it also employs statistical methods. However, in this

case, statistics is useful to estimate unknown variables of the analysed model, hence deriving desirable properties. The quantitative variables estimation is also achieved by performing Monte-Carlo simulations, except this time studies tend to analyse more closely the behaviours displayed by the agents [17]. In fact, most of this research [33, 16, 4] bases its results on Agent-Based Simulations (ABS), which is employed to stress test the actual smart contracts implementations being executed on a *"custom-built Ethereum virtual machine that is written in C++"* [33]. This research direction, although suggesting promising results, is not ultimately supported by strong statistical guarantees. This is mainly because the number of Monte Carlo simulations performed to run their analyses is arbitrary chosen and not backed by a formal justification [33, 26]. Nonetheless, a work relevant to this thesis is the analysis conducted by Kao et al. [33] on the Compound protocol scalability in face of high stock market prices volatility. Similarly to this thesis's one, their analysis models the prices by the use of the geometric Brownian motion as defined in Section 2.4.1. However, their data collection and analysis methodologies are very different. In fact, they do not sample entire historical periods as illustrated in Section 4.2 for estimating prices volatility. Contrarily, they simply evaluate the minimum and maximum volatility values ever observed and instantiate the GBM for different prices volatilities so to simulate several market environments. Finally, the prices model in Section 4.2 has been mostly inspired by Gudgeon et al.'s work [26]. Similarly to [33], they stress-test an LP model, not a specific implementation, by using the same price model explained in Section 4.2. Nonetheless, a remarkable difference is that they instantiate the predictions of the collateral and loan assets pairs with three different correlation parameters. This thesis, instead, assumes predictions of prices pairs to be strongly negatively correlated ($\rho = -1$), in order to simulate the worst-case scenario. Additionally, it reproduced Gudgeon et al.'s environment by using historical data of three different real cryptoassets on the market.

Therefore, this thesis can be viewed as a development of both the aforementioned research directions as it offers:

1. an accurate LP simulator which can support both the development of attacks and novel game-theoretic properties (Section 5.1);
2. a model checker capable of conducting simple reachability analysis and verifying whether LTL properties hold of specific system configurations (Section 5.2);
3. a tool for statistical analysis backed by the distributed and highly efficient MultiVeStA engine (Section 5.3).

Additionally, the thesis shows that under the scenarios developed in Section 4 and the assumptions explained in Section 5.3, $C_{\min} = 1.5$ and $r_{\text{liq}} = 1.1$ is the optimal parameters' choice to instantiate the model.

6.2 Limitations and mitigations

Perhaps, the most evident limitation of this work is that reachability analysis and LTL model checking are only capable of proving properties for configurations resulting in

finite computations. However, the LP model represents highly concurrent systems whose state space [6] is infinite. As a consequence, reachability analysis and LTL model checking are only useful to either confute invariant properties or explore the state space generated by some "interesting" initial states. This issue has been partially mitigated by enabling the initial simulator to conduct statistical analyses, Section 4.3. That allows to perform several bounded simulations of the model and conclude its properties based on statistical means, as explained in Section 2.3.3.

With regards to this limitation, it is convenient to argue in favour of the selection of the Maude system in order to specify and verify LP. This choice is motivated by at least three reasons.

1. The Maude language expressive power and its outstanding flexibility in modelling highly non-deterministic systems make Maude a suitable choice for specifying LP, as shown in Section 3.

2. Maude formal verification environment enables to conduct a variety of formal verification analyses on the specified systems, as illustrated in Section 5.

3. The Maude system is a free software project, distributed under the GNU-GPLv2.0. This made possible to adapt its source code as necessary in order to implement the tool. Listings A.2 and A.7 are the only files containing modules part of the Maude native libraries which have been modified and integrated in the tool.

Additionally, it could be argue that the LP simulator employed for statistical analysis, as defined in Section 4.3, is not consistent with the predictions of the prices in Section 4.2. This is mainly due to the fact that there is a one-to-one relation between a price update and a liquidation action. Furthermore, the price model is instantiated based on daily closing prices, which implicitly implies that predictions should also represent the daily evolution of a cryptoasset's closing prices. As a result, the current system evolves as if only one liquidation per day occurred. This does not seem to generate unexpected consequences on the statistical analysis's results explained in Section 5.3, but it might affect subsequent analyses. With respect to this, a simple correction of the issue would consist of a minimal modification to the logic of the model in Figure 4.4 so that the price update occurs only after the daily average number of liquidations has been performed.

6.3 Future research

Future research supported by the developed tool encompasses various areas of particular interest. First, Sections 5.1 and 5.2 showed how the tool is capable of simulating the LP model and verifying LTL properties for some initial configuration of LP. Consequently, a possible future development could be the formalisation of new attacks and properties of the model.

Secondly, the LP simulator used for statistical analysis could also be extended to find an answer to at least two more research questions.

1. Study the model resistance to illiquidity, as suggested by [33].
2. Investigate the behaviour of multi-pools configurations, each offering different market opportunities[86] to agents, as proposed by [55] and partially developed in [52].

The first research question could be addressed by implementing a heuristic for agents redeeming theirs collateral assets in a market environment promoting liquidations. The second problem can be handled by simply extending the simulator configuration in Listing 3.6, so that it contains multiple pools and integrating the interest rate models[87]. Additionally, for the second research problem multiple behaviours for a depositor might be developed. Two simple examples are risk-taking depositor, willing to invest in the platform even in adverse market conditions, or risk-adverse, depositing only when objectively convenient.

[86] In terms of prices and interest rates.
[87] For instance those in [27].

References

[1] Aave, S. Aave markets - webpage. https://aave.com/, 2021.

[2] Abdellatif, T., and Brousmiche, K.-L. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (2018), IEEE, pp. 1–5.

[3] Ammous, S. Blockchain technology: What is it good for? https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2832751, 2016.

[4] Angeris, G., Kao, H.-T., Chiang, R., Noyes, C., and Chitra, T. An analysis of uniswap markets. *Cryptoeconomic Systems Journal* (2019).

[5] Bai, X., Cheng, Z., Duan, Z., and Hu, K. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications* (2018), pp. 322–326.

[6] Baier, C., and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

[7] Bartoletti, M., Chiang, J. H.-y., and Lluch-Lafuente, A. Sok: Lending pools in decentralized finance. *arXiv preprint arXiv:2012.13230* (2020).

[8] Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J. Rigorous component-based system design using the bip framework. *IEEE software 28*, 3 (2011), 41–48.

[9] Bernardi, T., Dor, N., Fedotov, A., Grossman, S., Immerman, N., Jackson, D., Nutz, A., Oppenheim, L., Pistiner, O., Rinetzky, N., et al. Wip: Finding bugs automatically in smart contracts with parameterized invariants. https://groups.csail.mit.edu/sdg/pubs/2020/sbc2020.pdf, 2020.

[10] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. Bounded model checking. *Handbook of satisfiability 185*, 99 (2009), 457–481.

[11] Bigi, G., Bracciali, A., Meacci, G., and Tuosto, E. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*. Springer, 2015, pp. 142–161.

[12] Boado, E. Aave whitepaper. https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf, 2020. Accessed on 26.02.2021 - commit aeded1520c667e59a564cf69f33a6e489b2fe489.

[13] Boado, E., and Aave, S. Aave protocol maximum liquidate threshold. https://github.com/aave/aave-protocol/blob/1ff8418eb5c73ce233ac44bfb7541d07828b273f/contracts/lendingpool/LendingPoolLiquidationManager.sol#L181, 2021.

[14] Burkhardt, D., Werling, M., and Lasi, H. Distributed ledger. In *2018 IEEE international conference on engineering, technology and innovation (ICE/ITMC)* (2018), IEEE, pp. 1–9.

[15] Buterin, V. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2013. Accessed on 24.02.2021.

[16] Chitra, T., and Evans, A. Why stake when you can borrow? *arXiv e-prints* (2020), arXiv–2006.

[17] Chitra, T., Quaintance, M., Haber, S., and Martino, W. Agent-based simulations of blockchain protocols illustrated via kadena's chainweb. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2019), IEEE, pp. 386–395.

[18] Clarke, E. M. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science* (1997), Springer, pp. 54–56.

[19] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Rubio, R., and Talcott, C. Maude manual (version 3.0). Tech. rep., Technical report, SRI International Computer Science Laboratory, 2020.

[20] Compound Labs, I. Compound markets - webpage. https://compound.finance/markets, 2021.

[21] Dmouj, A. Stock price modelling: Theory and practice. *Masters Degree Thesis, Vrije Universiteit* (2006).

[22] Eker, S. Maude-help mail list - higher order functions. https://lists.cs.illinois.edu/lists/arc/maude-help/2006-01/msg00005.html, 2006.

[23] Eker, S., Meseguer, J., and Sridharanarayanan, A. The maude ltl model checker. *Electronic Notes in Theoretical Computer Science 71* (2004), 162–187.

[24] El Ioini, N., and Pahl, C. A review of distributed ledger technologies. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (2018), Springer, pp. 277–288.

[25] Entriken, W. Introduction to smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/, 2020. Accessed on 27.02.2021.

[26] Gudgeon, L., Perez, D., Harz, D., Livshits, B., and Gervais, A. The decentralized financial crisis. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)* (2020), IEEE, pp. 1–15.

[27] Gudgeon, L., Werner, S., Perez, D., and Knottenbelt, W. J. Defi protocols for loanable funds: Interest rates, liquidity and market efficiency. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (2020), pp. 92–112.

[28] Hendrix, J., Meseguer, J., and Sasse, R. Maude itp 2.0 tutorial. Tech. rep., Technical report, Technical report, University of Illinois at Urbana-Champaign, 2008.

[29] Holzmann, G. J. The model checker spin. *IEEE Transactions on software engineering 23*, 5 (1997), 279–295.

[30] Holzmann, G. J. Explicit-state model checking. In *Handbook of Model Checking*. Springer, 2018, pp. 153–171.

[31] Hull, J. C. *Options futures and other derivatives*. Pearson Education India, 2003.

[32] Jeffrey, G. Compound price oracle attack. https://news.bitcoin.com/100-million-liquidated-on-defi-protocol-compound-following-oracle-exploit/, 2020.

[33] Kao, H.-T., Chitra, T., Chiang, R., and Morrow, J. An analysis of the market risk to participants in the compound protocol. In *Third International Symposium on Foundations and Applications of Blockchains* (2020).

[34] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *LNCS*, Springer, pp. 585–591.

[35] Leshner, R., and Hayes, G. Compound: The money market protocol. https://compound.finance/documents/Compound.Whitepaper.v04.pdf, 2019.

[36] Liu, J., and Liu, Z. A survey on security verification of blockchain smart contracts. *IEEE Access 7* (2019), 77894–77904.

[37] Meseguer, J. Membership algebra as a logical framework for equational specification. In *International Workshop on Algebraic Development Techniques* (1997), Springer, pp. 18–61.

[38] Mirelli, M. A maude simulator for lending pools. https://github.com/MMirelli/maude-lp, 2021. Accessed on 19.06.2021 - commit 2dae39b035938f5f9791040c53121fb473b4b7dd.

[39] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Satoshi Nakamoto Institute, 2008.

[40] Nofer, M., Gomber, P., Hinz, O., and Schiereck, D. Blockchain. *Business & Information Systems Engineering 59*, 3 (2017), 183–187.

[41] Perez, D., Werner, S. M., Xu, J., and Livshits, B. Liquidations: Defi on a knife-edge. *arXiv preprint arXiv:2009.13235* (2020).

[42] Peterins, E., Flatow, J., Hayes, G., Wolff, M., and Greenberg, A. Compound protocol maximum liquidate threshold. https://github.com/compound-finance/compound-protocol/blob/4e99ea3a64ab4f1bdf9c07c7a1bf325db09ab809/scenario/src/Event/ComptrollerEvent.ts#L170, 2021.

[43] Plotkin, G. D. *A structural approach to operational semantics*. Aarhus university, 1981.

[44] Pulse. Defi pulse - webpage. https://defipulse.com, 2021. Accessed on 07.06.2021.

[45] Qin, K., Zhou, L., Livshits, B., and Gervais, A. Attacking the defi ecosystem with flash loans for fun and profit. *arXiv preprint arXiv:2003.03810* (2020).

[46] Rauchs, M., Glidden, A., Gordon, B., Pieters, G. C., Recanatini, M., Rostand, F., Vagneur, K., and Zhang, B. Z. Distributed ledger technology systems: A conceptual framework. *Available at SSRN 3230013* (2018).

[47] Scholten, O. J., Hughes, N. G. J., Deterding, S., Drachen, A., Walker, J. A., and Zendle, D. Ethereum crypto-games: Mechanics, prevalence, and gambling similarities. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play* (2019), pp. 379–389.

[48] Sebastio, S., and Vandin, A. Multivesta: Statistical model checking for discrete event simulators. Tech. rep., IMT Institute for Advanced Studies Lucca, 2013.

[49] SRI International, I. Maude v3 - release. http://maude.cs.illinois.edu/w/index.php?title=Maude_download_and_installation&oldid=252, 2019.

[50] Szabo, N. Introduction to smart contracts. https://firstmonday.org/ojs/index.php/fm/article/download/548/469, 1997. Accessed on 27.02.2021.

[51] Ter Beek, M. H., Legay, A., Lafuente, A. L., and Vandin, A. A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Transactions on Software Engineering 46*, 3 (2020), 321–345.

[52] Tolmach, P., Li, Y., Lin, S.-W., and Liu, Y. Formal analysis of composable defi protocols. *arXiv preprint arXiv:2103.00540* (2021).

[53] Vandin, A., Giachini, D., Lamperti, F., and Chiaromonte, F. Automated and distributed statistical analysis of economic agent-based models. *arXiv preprint arXiv:2102.05405* (2021).

[54] Wackerow, P., and Rhechler. Decentralized finance (defi) - webpage. https://ethereum.org/en/defi/, 2021. Accessed on 02.06.2021.

[55] Werner, S. M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., and Knottenbelt, W. J. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778* (2021).

[56] Zetzsche, D. A., Arner, D. W., and Buckley, R. P. Decentralized finance. *Journal of Financial Regulation 6*, 2 (2020), 172–203.

[57] Zheng, Z., Xie, S., Dai, H., Chen, X., and Wang, H. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)* (2017), IEEE, pp. 557–564.

[58] Zhou, L., Qin, K., Cully, A., Livshits, B., and Gervais, A. On the just-in-time discovery of profit-generating transactions in defi protocols. *arXiv preprint arXiv:2103.02228* (2021).

[59] Ziechmann, K. Ethereum virtual machine. https://ethereum.org/en/developers/docs/evm/, 2021. Accessed on 27.02.2021.

A Maude specifications

A.1 Abstract datatypes

This listing specifies the main abstract data types, utilised in the rest of the LP model specifications. The main specified modules are:

`SUBSET`	specifies sets of tokens types;
`MAP`	models maps (or dictionaries) in Maude and it is mainly defined accordingly to the native Maude library `prelude.maude`, as later discussed;
`MAP-TH`	implements functions (in the mathematical sense), binding target sorts to the ones in `FUNCTION`;
`FUNCTION`	implements functions (in the mathematical sense), containing the logic for computing domain and codomain;
`HO-FILTER`	implements the filtering using higher order functions.

The fundamental difference between `Map{X,Y}` used in LP specification and the original is shown in Listing A.1, illustrating the declaration of the original `Map{X,Y}`'s constructor. Notably, this operator differs than the LP one, Listing A.2 Line 189, for the two reasons detailed below.

1. The former constructs a `Map` from `Map` terms only, whereas the latter constructs a `Map` term incrementally from `Entries`.
2. The former does not utilise kinds, while the latter does.

These modifications to the original `Map` have been made, in order to develop the type system of tokens' `Maps`, which is not discussed in this thesis.

```
op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
   [ctor assoc comm id: empty prec 121 format (d r os d)] .
```

Listing A.1: Maude original definition of `Map{X,Y}`, as per `prelude.maude`.

```
    ⋮              ⋮              ⋮

fmod SUBSET{X :: TRIV, Y :: TRIV} is
   including SET{Y} .
   sorts NeSet{X} Set{X} .

   subsort X$Elt < Y$Elt .
   subsort NeSet{X} < NeSet{Y} .
   subsort Set{X} < Set{Y} .

   subsort X$Elt < NeSet{X} < Set{X} .

   op _,_ : Set{X} Y$Elt -> Set{Y}
 [ctor id: empty assoc comm prec 121 format (d r os d)] .

   op _,_ : Set{X} Set{X} -> Set{X} [ctor ditto] .
   op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .

endfm
    ⋮              ⋮              ⋮

endv

fmod MAP{X :: TRIV, Y :: TRIV} is
  including BOOL .
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor prec 50] .
  op emptyM : -> Map{X,Y} [ctor] .
  --- original
  ---  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
  ---   [ctor assoc comm id: empty prec 121 format (d r os d)] .

  --- the Entry tries to solve the collapse problem
  --- described in 20.3.6
  --- the [Map] solves the problem introduced by tokens
  --- subtypying
  op _;_ : [Map{X,Y}] Entry{X,Y} -> [Map{X,Y}]
   [ctor assoc comm id: emptyM prec 51 format (d r os d)] .
  op undefined : -> [Y$Elt] [ctor] .

  var D : X$Elt .
  vars R R' : Y$Elt .
  var M : Map{X,Y} .

  mb (M:Map{X,Y}); (E:Entry{X,Y}) : Map{X,Y} .

  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
```

```
    eq insert(D, R, (M ; D |-> R')) =
       if $hasMapping(M, D) then insert(D, R, M)
       else (M ; D |-> R)
       fi .
    eq insert(D, R, M) = (M ; D |-> R) [owise] .

    op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 21] .
                                        --- _+_ is 22
    eq (M ; D |-> R)[D] =
       if $hasMapping(M, D) then undefined
       else R
       fi .
    eq M[D] = undefined [owise] .

    op $hasMapping : Map{X,Y} X$Elt -> Bool .
    eq $hasMapping((M ; D |-> R), D) = true .
    eq $hasMapping(M, D) = false [owise] .
  endfm
    ⋮              ⋮              ⋮

  fth MAP-THEORY is
    sorts M E DT CT D C .

    subsort E < M .
    op _|->_ : DT CT -> E [ctor] .
    op _;_ : [M] E -> [M] [ctor id: emptyM assoc comm prec
                                    121 format (d r os d)] .
    op emptyM : -> M .

    subsort DT < D .
    subsort CT < C .
    op emptyC : -> C .
    op _,_ : D DT -> D
  [ctor assoc comm prec 121 format (d r os d)] .

    op emptyD : -> D .
    op _,_ : C CT -> C
  [ctor assoc comm prec 121 format (d r os d)] .
  endfth
    ⋮              ⋮              ⋮

  fmod FUNCTION{MT :: MAP-THEORY} is
    var map : MT$M .
    var domElt : MT$DT .
    var codElt : MT$CT .
    var domSet : MT$D .
    var codSet : MT$C .

    op dom : MT$M -> MT$D .
    eq dom(map) = $dom(map, emptyD) .

    op $dom : MT$M MT$D -> MT$D .
```

```
    eq $dom((map ; domElt |-> codElt), domSet) =
                              $dom(map, (domSet, domElt)) .
    eq $dom((domElt |-> codElt), domSet) = domSet, domElt .
    eq $dom(emptyM, domSet) = domSet .

    op cod : MT$M -> MT$C .
    eq cod(map) = $cod(map, emptyC) .

    op $cod : MT$M MT$C -> MT$C .
    eq $cod((map ; domElt |-> codElt), codSet) =
                              $cod(map, (codSet, codElt)) .
    eq $cod((domElt |-> codElt), codSet) = codSet, codElt .
    eq $cod(emptyM, codSet) = codSet .
endfm

view FUNCTION{M :: MAP-THEORY} from TRIV to FUNCTION{M} is
    sort Elt to M$M .
endv

fmod AP{X :: TRIV, Y :: TRIV} is
sort Func{X, Y} .
op _[_] : Func{X, Y} X$Elt -> Y$Elt [prec 17] .
endfm

fmod AP-BIN{X :: TRIV, Y :: TRIV, Z :: TRIV} is
sort Func{X, Y, Z} .
op _[_,_] : Func{X, Y, Z} X$Elt Y$Elt -> Z$Elt [prec 17] .
endfm

fmod HO1{X :: TRIV} is
inc LIST{X} .
endfm

fmod HO2{X :: TRIV, Y :: TRIV} is
inc HO1{X} .

inc LIST{Y} .
endfm

fmod HO-FILTER{X :: TRIV} is
including HO1{X} .
including BOOL .
inc AP{X, Bool} .

var E : X$Elt .
var L : List{X} .
var P : Func{X, Bool} . --- predicate

op filter : List{X} Func{X, Bool} -> List{X} .
eq filter(nil, P) = nil .
ceq filter(E | L, P) = E | filter(L, P)
    if P[E] .
eq filter(E | L, P) = filter(L, P) [owise] .
```

```
endfm
    ⋮              ⋮              ⋮
```

Listing A.2: `abstract-datatypes.maude` - Maude specifications of parameterised data types

A.2 Tokens

This listing specifies the sorts and operators implementing token types and sets of token types.

```
fmod TOKENS is
    --- token types
    including BOOL .
    including INT .

    --- native token is unique (not a set) and it's free (it
    --- can be transferred)
    sort Token Free-Token Minted-Token .
    subsort Minted-Token Free-Token < Token .

    sort Native-Token .
    subsort Native-Token < Free-Token .

    op tauError : -> [Token] .
    op dummyTau : Nat -> Free-Token .

    --- _[_] has prec 21
    op tau(_) : Nat -> Free-Token
                      [ctor prec 19 format (y! d d d o)] .
    --- _[_] has prec 21
    op _' : Free-Token -> Minted-Token
                      [ctor prec 20 format (d b! o)] .
    op ethtau : -> Native-Token .
endfm

view Token from TRIV to TOKENS is
  sort Elt to Token .
endv

view Free-Token from TRIV to TOKENS is
  sort Elt to Free-Token .
endv

view Native-Token from TRIV to TOKENS is
  sort Elt to Native-Token .
endv

view Minted-Token from TRIV to TOKENS is
  sort Elt to Minted-Token .
```

```
endv

fmod TOKEN-SETS is
   including TOKENS .
   including SUBSET{Free-Token, Token} .
   including SUBSET{Minted-Token, Token} .

   op freeTokenSet : -> Set{Free-Token} .
   eq freeTokenSet = tau(0), tau(1) .

   op mintableTokenSet : -> Set{Minted-Token} .
   eq mintableTokenSet = tau(0)', tau(1)' .

   op allTokenSet : -> Set{Token} .
   eq allTokenSet = ethtau, freeTokenSet, mintableTokenSet .

   vars anyTau : Token .
   vars anyTauSet curFTok : Set{Token} .

   op filterFree : Set{Token} -> Set{Free-Token} .
   eq filterFree(anyTauSet) =
                  $filterFree(anyTauSet, empty) .

   op $filterFree : Set{Token} Set{Free-Token} ->
                               Set{Free-Token} .
   eq $filterFree(empty, curFTok) = curFTok .
   eq $filterFree((anyTauSet, anyTau), curFTok) =
               if anyTau :: Free-Token
               then
                  $filterFree(anyTauSet, (curFTok, anyTau))
               else
                  $filterFree(anyTauSet, curFTok)
               fi .

   op filterMinted : Set{Token} -> Set{Minted-Token} .
   eq filterMinted(anyTauSet) =
                    anyTauSet \ filterFree(anyTauSet) .

    op getFirst : NeSet{Token} -> Token .
    eq getFirst((anyTauSet, anyTau)) = anyTau .

endfm

fmod TOKEN-LISTS is
   including TOKENS .
   including SUBLIST{Free-Token, Token} .
   including SUBLIST{Minted-Token, Token} .

   op freeTokenList : -> List{Free-Token} .
   eq freeTokenList = tau(0) | tau(1) .

   op mintableTokenList : -> List{Minted-Token} .
   eq mintableTokenList = tau(0)' | tau(1)' .
```

```
    op allTokenList : -> List{Token} .
    eq allTokenList = ethtau | freeTokenList |
                               mintableTokenList .

endfm

fmod TOKEN-MAP is
    including TOKEN-SETS .
    including MAP{Token, Float0+} .
    including SET{Float0+} .
endfm

view PiFFun from MAP-THEORY
                to TOKEN-MAP is
    sort M to Map{Token, Float0+} .
    sort E to Entry{Token, Float0+} .

    sort DT to Token .
    sort CT to Float0+ .

    sort D to Set{Token} .
    sort C to Set{Float0+} .

    op emptyD to empty .
    op emptyC to empty .
endv

fmod SUBTOKEN-FUNCTIONS is
    --- importing agents
    including AGENT-ID-SETS .

    including FUNCTION{PiFFun} .
    including SUBMAP{Free-Token, Float0+, PiFFun} .
    including SUBMAP{Minted-Token, Float0+, PiFFun} .

    including MAP{Free-Token, Pair{Token, Float0+}} .
    --- pi_loan & pi_fund/free & price
    ------  (Free-Token -> Float0+) < (Token -> Float0+)
    including MAP{Agent-Id, Map{Token, Float0+}} .

    --- For increased type accuracy use this:
    --- including MAP{Agent-Id, Map{Free-Token, Float0+}} .
    including SET{Pair{Token, Float0+}} .
    including SET{Map{Token, Float0+}} .
endfm


view PiLFun from MAP-THEORY
                to SUBTOKEN-FUNCTIONS is
    sort M to Map{Agent-Id, Map{Token, Float0+}} .
    sort E to Entry{Agent-Id, Map{Token, Float0+}} .

```

```
    sort DT to Agent-Id .
    sort CT to Map{Token, Float0+} .

    sort D to Set{Agent-Id} .
    sort C to Set{Map{Token, Float0+}} .

    op emptyD to empty .
    op emptyC to empty .
endv

view PiMFun from MAP-THEORY
                to SUBTOKEN-FUNCTIONS is
    sort M to Map{Free-Token,
                  Pair{Token, Float0+}} .
    sort E to Entry{Free-Token,
                  Pair{Token, Float0+}} .

    sort DT to Token .
    sort CT to Pair{Token, Float0+} .

    sort D to Set{Token} .
    sort C to Set{Pair{Token, Float0+}} .

    op emptyD to empty .
    op emptyC to empty .
endv

fmod HO-EXISTS-TEST is
    extending HO-EXISTS{Token} .

    var tau : Token .
    op isFree : -> Func{Token, Bool} .
    eq isFree[tau] = tau :: Free-Token .
endfm

--- NOTE: UNUSED
view TokenFun from MAP-THEORY to SUBTOKEN-FUNCTIONS is
    sort M to Map{Token, Float0+} .

    sort E to Entry{Token, Float0+} .

    sort DT to Token .
    sort CT to Float0+ .

    sort D to Set{Token} .
    sort C to Set{Float0+} .

    op emptyD to empty .
    op emptyC to empty .
endv
```

Listing A.3: `tokens.maude` - Maude specifications of tokens

A.3 Wallets

This listing specifies the sorts and operators implementing the wallets function σ. The main specified modules are:

`WALLET`	defines the syntax and APIs offered by the wallets function, σ;
`PI-FUND`	defines the syntax and APIs offered by π_f.

```
fmod WALLET is
    including MAP{Token, Float0+} .

    var bal : Map{Token, Float0+} .
    var tau : Token .
    var v v' : Float0+ .

    op _+_:_ : Map{Token, Float0+} Float0+ Token ->
                        Map{Token, Float0+} [prec 22] .
    ceq bal + v : tau = insert(tau, (bal[tau]) + v, bal)
          if bal[tau] =/= undefined .
    eq bal + v : tau = insert(tau, v, bal) [owise] .

    op _-_:_ : Map{Token, Float0+} Float0+ Token ->
                       Map{Token, Float0+} [prec 22] .
    ceq bal - v : tau = insert(tau, (bal[tau]) - v, bal)
          if bal[tau] =/= (undefined).Float0+ /\
                           (bal[tau] - v) : Float0+ .
    eq bal - v : tau = bal [owise] .

    op _*_:_ : Map{Token, Float0+} Float0+ Token ->
                        Map{Token, Float0+} [prec 21] .
    ceq bal * v : tau = insert(tau, (bal[tau]) * v, bal)
          if bal[tau] =/= undefined .
    eq bal * v : tau = bal [owise] .


    vars m0 m1 : Map{Token, Float0+} .
    op _*_ : Map{Token, Float0+} Map{Token, Float0+}
                        -> Map{Token, Float0+} [prec 22] .
    eq m0 * emptyM = m0 .
    eq m0 * (tau |-> v) = m0 * v : tau .
    eq m0 * (m1 ; (tau |-> v)) = (m0 * v : tau) * m1 .

endfm

fmod PI-FUND is
    including FUNCTION{PiFFun} .
```

```
41      --- tells if 2 maps, with equal domain, forall token in
        --- their domains are equals with a difference of at
        --- most eps
        var eps : Float0+ .
        var m m' : Map{Token, Float0+} .
46      var tau : Token .
        var tokS : Set{Token} .
        var v v' : Float0+ .

        op _~=_:_ : Map{Token, Float0+}
51                  Float0+ Float0+ -> Bool .
        eq emptyM ~= v' : eps = true .
        eq (m ; tau |-> v) ~= v' : eps =
             if abs(v - v') <= eps then
                m ~= v' : eps
56           else
                false
             fi .

        op _~=_:_ : Map{Token, Float0+}
61                  Map{Token, Float0+}  Float0+
                                              -> Bool .
        eq m ~= m' : eps =
            if (dom(m) == dom(m')) then
                m ~=[dom(m)] m' : eps
66          else
                false
            fi .


71      op _~=[_]_:_ : Map{Token, Float0+} Set{Token}
                       Map{Token, Float0+}  Float0+
                                                 -> Bool .
        eq m ~=[empty] m' : eps = true .
        eq m ~=[tau] m' : eps = abs(m[tau] - m'[tau]) <= eps .
76      eq m ~=[tokS, tau] m' : eps =
                if abs(m[tau] - m'[tau]) <= eps
                then
                    m ~=[tokS] m' : eps
                else
81                  false
                fi .

        op _~<[_]_:_ : Map{Token, Float0+} Set{Token}
                       Map{Token, Float0+}  Float0+
86                                             -> Bool .
        eq m ~<[empty] m' : eps = true .
        eq m ~<[tau] m' : eps = (m'[tau] - m[tau]) > eps .
        eq m ~<[tokS, tau] m' : eps =
                if (m'[tau] - m[tau]) > eps
91              then
                    m ~<[tokS] m' : eps
                else
```

```
                false
            fi .

    op _~<_:_ : Map{Token, Float0+}
                Map{Token, Float0+}  Float0+
                                          -> Bool .
    eq m ~< m' : eps =
        if (m' =/= emptyM) and-then
           ((dom(m) == empty) or-else
                    (dom(m) subset dom(m'))) then
            m ~<[dom(m)] m' : eps
        else
            false
        fi .
endfm

    ⋮             ⋮             ⋮
```

Listing A.4: `wallets.maude` - Maude specifications of agent wallets and π_f

A.4 Pools

This listing specifies the sorts and operators implementing the pool object π. The main specified modules are:

`LP`	defines the services offered by a pool π;
`BC{LP}`	defines the syntax of an LP state, comprising a pool π and a price function p.

```
    ⋮             ⋮             ⋮

fmod LP is
    including PI-LOAN + PI-MINT + PI-FUND .
    including WALLET .

    sort Pi .

    eq (undefined).Set{Float0+} = 0.0 .
    *** eq snd((undefined).Set{Pair{Token,Float0+}}) =
    ***                              0.0 .
    eq (undefined).Set{Map{Token,Float0+}} + v : tau =
                   tau |-> v .
    eq (undefined).Map{Token,Float0+}[tau] = 0.0 .
```

```
82      var pi : Pi .
        var pi_f : Map{Free-Token, Float0+} .
        --- For increased type accuracy use this:
        *** var pi_l : Map{Agent-Id, Map{Free-Token, Float0+}
        var pi_l : Map{Agent-Id, Map{Token, Float0+}} .
87      var pi_m : Map{Free-Token,
                           Pair{Token, Float0+}} .
        var v : Float0+ .
        var tau : Token .

92      vars fTau fTau' fTau'' : Free-Token .
        vars mTau : Minted-Token .
        var mintedPair : Pair{Token, Float0+} .
        var fTokenSet fTokenSet' : Set{Free-Token} .

97      op {fund:_, loan:_, mint:_} :
              Map{Free-Token, Float0+}
              Map{Agent-Id, Map{Token, Float0+}}
              Map{Free-Token, Pair{Token, Float0+}} -> Pi
                [prec 49 ctor] .
102           *** NOTE: pretty printing removed for multivesta
              *** format(nt++ nti d d nti d d nti d nt d) ] .

        op _.fund : Pi -> Map{Free-Token, Float0+}
                         [prec 20  format(d - d)] . --- 21 is []
107     eq ({fund: pi_f, loan: pi_l, mint: pi_m}).fund = pi_f .
        eq ({fund: emptyM, loan: pi_l, mint: pi_m}).fund =
                          (emptyM).Map{Free-Token, Float0+} .

        op _.loan : Pi -> Map{Agent-Id, Map{Token, Float0+}}
112                      [prec 20  format(d - d)] . --- 21 is []
        eq ({fund: emptyM, loan: pi_l, mint: pi_m}).loan = pi_l .
        eq ({fund: pi_f, loan: pi_l, mint: pi_m}).loan = pi_l .

        op _.mint : Pi ->
117           Map{Free-Token, Pair{Token, Float0+}}
                         [prec 20  format(d - d)] . --- 21 is []
        eq ({fund: emptyM, loan: pi_l, mint: pi_m}).mint = pi_m .
        eq ({fund: pi_f, loan: pi_l, mint: pi_m}).mint = pi_m .

122

        op emptyPi : -> Pi .
        eq emptyPi = {fund: emptyM, loan: emptyM, mint: emptyM} .

        mb {fund: (emptyM).Map{Token,Float0+},
127           loan: pi_l, mint: pi_m} : Pi .


        op isInitial : Pi -> Bool .
        eq isInitial(pi) = pi == (emptyPi) .
132
```

```
    --- Map{Agent-Id, Map{Token, Float0+}}
    op sum : Pi Free-Token -> Float0+ .
    eq sum( pi, fTau ) = $sum( (pi).loan, fTau,
137                                dom((pi).loan), 0.0) .

    var aSet : Set{Agent-Id} .
    var a : Agent-Id .

142 op $sum : Map{Agent-Id,Map{Token,Float0+}} Free-Token
                  Set{Agent-Id} Float0+ -> Float0+ .
    eq $sum(pi_l, fTau, (empty).Set{Agent-Id}, v) = v .
    eq $sum(pi_l, fTau, (aSet, a), v) =
       if (pi_l[a]) [fTau] =/= undefined then
147       $sum(pi_l, fTau, aSet, v + (pi_l[a]) [fTau])
       else
          $sum(pi_l, fTau, aSet, v)
       fi .

152 op _.ER_ : Pi Free-Token -> Float0+ [prec 30] .
                                        --- _*_ 31
    ceq (pi).ER(fTau) = ((pi).fund[fTau] + sum(pi, fTau)) /
                            snd((pi).mint[fTau])
        if (pi).fund[fTau] > 0.0 .
157 eq (pi).ER(fTau) = 1.0 [owise] .

    --- returns the minted token with respect to a
    --- single free token (i.e. fTau' for fTau free-token,
    --- for fTau = fst(\pi.mint(fTau'))
162 op minted : Pi Free-Token -> Minted-Token .
    eq minted(pi, fTau) =
          if (pi).mint == emptyM then
             empty
          else
167          $minted( (pi).mint, fTau)
          fi .

    --- auxiliary for minted(Pi Token)
    op $minted :
172   Map{Free-Token, Pair{Token, Float0+}}
      Free-Token ->
                     Minted-Token .
    eq $minted ( (fTau' |-> mintedPair), fTau) =
         if (fTau == fTau') then
177         fst(mintedPair)
         else
            empty
         fi .
    eq $minted ( (pi_m ; (fTau' |-> mintedPair)), fTau) =
182      if (fTau == fTau') then
            fst(mintedPair)
         else
            $minted(pi_m, fTau)
         fi .
```

```
187
     --- gets the free tokens (dom(pi_f))
     op (_).freeTokens : Pi -> Set{Free-Token} .
     eq (pi).freeTokens = dom((pi).fund) .

192  --- gets the minted tokens (cod(fst(pi_m)))
     op (_).mintedTokens : Pi -> Set{Minted-Token} .
     eq (pi).mintedTokens =
                  $mintedTokens(cod((pi).mint), empty) .

197  var mTokPairSet : Set{Pair{Token, Float0+}} .
     var tokenSet : Set{Token} .

     op $mintedTokens : Set{Pair{Token, Float0+}}
                        Set{Token} -> Set{Minted-Token} .
202  eq $mintedTokens(empty, tokenSet) = tokenSet .
     eq $mintedTokens((mTokPairSet, mintedPair), tokenSet) =
             $mintedTokens(mTokPairSet,
                           (tokenSet, fst(mintedPair))) .
     eq $mintedTokens((mintedPair), tokenSet) =
207                        (tokenSet, fst(mintedPair)) .

     --- underlying minted tokens function
     --- Assumption:
     op _.u_ : Pi Minted-Token -> Free-Token
212                   [prec 21] . --- _+_:_ prec is 22
     eq (pi).u(mTau) = if ( (pi).mint =/= emptyM )
                       then
                          $u( (pi).mint, mTau)
                       else
217                       empty
                       fi .

     --- auxiliary for u
     op $u :
222       Map{Free-Token, Pair{Token, Float0+}}
                Minted-Token -> Free-Token .
     eq $u( (pi_m ; (fTau |-> mintedPair)), mTau ) =
               if fst(mintedPair) == mTau then
                  fTau
227            else
                  $u(pi_m, mTau)
               fi .
endfm
     ⋮             ⋮             ⋮

--- Abstract Blockchain configuration module
283  fmod BC{X :: BC-ENV} is
     --- BC: BlockChain Configuration
     sort BC{X}  .
     op _|_ : X$Env X$Price -> BC{X} [prec 53] .
            --- NOTE: pretty printing removed for multivesta
```

```
288                    *** format (d bn++s ont n)] .
    endfm


    fmod BC-LP is
293     including BC{LP} .

        var pi : Pi .
        var price : Map{Token, Float0+} .

298     --- isInitial: true iff lending pools blockchain context
        --- is initial
        ---        i.e. \pi_i = empty \forall \pi_i in \pi.keys()
        op isInitial : BC{LP} -> Bool .
        eq isInitial (pi | price) = isInitial(pi) .
303
        op _.pi : BC{LP} -> Pi [prec 21] . --- 22 is .mint
        eq (pi | price).pi = pi .

        op _.price : BC{LP} -> Map{Token, Float0+}
308                             [prec 21] . --- 22 is .mint
        eq (pi | price).price = price .

    endfm
        ⋮               ⋮               ⋮
```

Listing A.5: `pools.maude` - Maude specifications of LPs main data structure: π

A.5 Configurations

This listing specifies the sorts and operators implementing a lending pool configuration Γ. The main specified modules are:

`CONFIGURATION`	models a generic Maude configuration and it is mainly defined accordingly to the native Maude library `prelude.maude`, as later discussed;
`LP-PARAMETERS`	defines the LP parameters $C_{\min}$, r_{liq} and *Maxliq*;
`BC-LP-MODEL`	defines the LP rules;
`BC-LP-MOD-MOD`	defines extra services for an LP configuration, used for the integration with MultiVeStA.

It is essential to note that Maude provides the implementation for a generic

`Configuration`[88]. However, the LP `Configuration` is slightly different and corresponds to the one shown in Listings 2.2 and A.7, Line 1. Listing A.6 illustrates the original constructor for a `Configuration` multiset, which is the only remarkable difference with the LP `Configuration`, in Listing A.7, Line 16. As it is evident, the change only affects the operator input sorts, being `[Configuration]` in the operator used by LP specifications and `Configuration` in the original. This is motivated by the fact that `[Configuration]` identifies the kind of sort `Configuration`. Consequently, the adopted operator allows to define well-formed terms for a `Configuration` by simply defining a conditional membership equation statement as the one shown in Listing A.7, Line 86. From the listing, a well-formed `Configuration` is defined to be one satisfying the condition `isLPStateUnique`. Intuitively, this function checks that only one term `(pi | p):BC{LP}` is contained in the input `Configuration`.

```
op __ : Configuration Configuration -> Configuration
      [prec 53 ctor config assoc comm id: none] .
```

Listing A.6: Maude original definition of `Configuration`, as per `prelude.maude`.

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet  [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
                          [ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Portal Configuration .
  subsort Object Msg Portal < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object
                [ctor object].
           *** NOTE: pretty printing removed for multivesta
           *** format(n d d d d d d d)] .
  op none : -> Configuration [ctor] .
  op errorConfig : -> [Configuration] .
  op __ : [Configuration] [Configuration] -> [Configuration]
                 [prec 53 ctor config assoc comm id: none] .
  op <> : -> Portal [ctor] .


  var oid : Oid .
  var cid : Cid .
  var atts : AttributeSet .
  var obj : Object .
  vars msg msg' : Msg .
  vars gamma gamma' : [Configuration] .

  op unknownClass : ->  Cid [ctor] .
  op _.cid : Object -> Cid [prec 50] .
```

[88] Introduced in Section 2.2.2.

```
30    eq (< oid : cid | atts >).cid  = cid .
      eq (obj).cid = unknownClass .

      op unknownObjId : ->  Oid [ctor] .
      op _.oid : Object -> Oid [prec 50] .
35    eq (< oid : cid | atts >).oid  = oid .
      eq (obj).oid = unknownObjId .

      op _in_ : Msg [Configuration] -> Bool .
      eq msg in (none) = false .
40    eq msg in (gamma obj) = msg in (gamma) .
      eq msg in (gamma msg) = true .
      eq msg in (gamma msg') = msg in (gamma) .
   endm
       ⋮              ⋮              ⋮

   mod BC-LP-CONFIG-COMPONENTS is
53       including BC-LP .
         including PAIR{Agent-Id, Float0+} .
         including CONFIGURATION .

   ---------------------- Pool ---------------------------------
58       subsort BC{LP} < Object .

   ------------------- Coll object in config -------------
         op C : Int -> Oid .
         op Coll : -> Cid [ctor] .
63       op *_ : Pair{Agent-Id, Float0+} -> Attribute
                     [ctor gather (&)] .

   ----------------------- BC Round ---------------------------
         op R : Int -> Oid .
68       op Round : -> Cid [ctor] .
         op *_ : Int -> Attribute
                     [ctor gather (&)] .

   ---------------------- closedConfiguration -----------------
73       var wf-gamma : Configuration .
         sort closedConfiguration .
         op [_] : Configuration -> closedConfiguration [prec 50] .

         op (_).config : closedConfiguration -> Configuration .
78       eq ([wf-gamma]).config = wf-gamma .

         vars gamma gamma' : [Configuration] .
         var sigma : Map{Token, Float0+} .
         var lp lp' : BC{LP} .
83       var N : Nat .

         --- LP State uniqueness check
         cmb gamma : Configuration if isLPStateUnique(gamma, 0)
                                                    == 1 .
```

```
88
     op isLPStateUnique : [Configuration] Nat -> Nat .

     eq isLPStateUnique(gamma lp, N) =
                              isLPStateUnique(gamma, N + 1) .
93   eq isLPStateUnique(lp, N) = N + 1 .
     eq isLPStateUnique(none, N) = N .

     eq isLPStateUnique(none gamma, N) = N .
     eq isLPStateUnique(gamma gamma', N) =
98                                 isLPStateUnique(gamma, N) .
endm
     ⋮              ⋮              ⋮

mod LP-PARAMETERS is
     including CONFIGURATION .
     including FLOAT0+ .
114  including INT .

     op P : Int -> Oid .
     op LiqParams : -> Cid [ctor] .

119  sort Param .
     ops CMin Rliq : Float0+ -> Param .
     subsort Param < Attribute .

     sort Params .
124  subsort Params < Object .

     var n : Int .
     vars cminV rliqV : Float0+ .
     var params : AttributeSet .
129  mb < P(n) : LiqParams | CMin(cminV),
                             Rliq(rliqV) > : Params .

     var paramObj : Params .
     op (_).CMin : Params -> Float0+ .
134  eq (< P(n) : LiqParams | CMin(cminV),
                              Rliq(rliqV) >).CMin = cminV .

     op (_).Rliq : Params -> Float0+ .
     eq (< P(n) : LiqParams | CMin(cminV),
139                           Rliq(rliqV) >).Rliq = rliqV .

     op Maxliq : -> Float0+ .
     eq Maxliq = 0.5 .

144  --- used as placeholder for undefined collateralization
     op - : -> Float0+ .
endm

mod AGENT-STATE is
```

```
149     including CONFIGURATION .
        including MAP{Token, Float0+} .
        including AGENT-ID .
        including LP-PARAMETERS .

154 ---------------------- Agent-State ------------------------
        sort Agent-State .
        subsort Agent-State  < Cid .
        op noState : -> Agent-State [ctor] .

159     var s : Float0+ .
        op LIQ-Speed : Float0+ -> [Agent-State] [ctor] .

        cmb LIQ-Speed(s) : Agent-State if s <= Maxliq .

164     op LIQ-MaxSpeed : -> [Agent-State] [ctor] .
        eq LIQ-MaxSpeed = LIQ-Speed(Maxliq) .

        op *_ : Map{Token, Float0+} -> Attribute
                    [ctor gather (&)] .
169
        var A0 : Agent-Id .
        var b : Map{Token, Float0+} .

        var agState : Agent-State .
174
        sort Agent .
        subsort Agent < Object .
        subsort Agent-Id < Oid .
        mb < A0 : agState | * b > : Agent .
179
-------- Agent-State = (Agent-Type, Agent-Action) --------
        sorts Agent-Type Agent-Action .
endm
        ⋮                ⋮                ⋮

mod BC-LP-CONFIGURATION is
202     including BC-LP-CONFIG-COMPONENTS .
        including AGENT-STATE .
        including LP-PARAMETERS .

------------------ short deconstructors -------------
207     var gamma : [Configuration] .
        var obj : Object .
        op objectNotFound : -> [Object] .

        op _.lp : [Configuration] -> BC{LP} .
212     eq (none).lp = objectNotFound .
        ceq ((gamma obj)).lp = obj if obj :: BC{LP} .
        eq ((gamma obj)).lp = (gamma).lp [owise] .

        var wf-gamma : Configuration .
```

```
217     op (_).mintedTokens : Configuration ->
                                  Set{Minted-Token} .
        eq (wf-gamma).mintedTokens =
                             (((wf-gamma).lp).pi).mintedTokens .

222     op (_).pi.f : Configuration ->
            Map{Free-Token, Float0+} .
        eq (wf-gamma).pi.f = (((wf-gamma).lp).pi).fund .

        op (_).pi.l : Configuration ->
227         Map{Agent-Id, Map{Token, Float0+}} .
        eq (wf-gamma).pi.l = (((wf-gamma).lp).pi).loan .

        op (_).pi.m : Configuration ->
            Map{Free-Token, Pair{Token, Float0+}} .
232     eq (wf-gamma).pi.m = (((wf-gamma).lp).pi).mint .

        var tau' : Minted-Token .

        op (_).u(_) : Configuration Minted-Token ->
237                                           Free-Token .
        eq (wf-gamma).u(tau') = (((wf-gamma).lp).pi).u(tau') .


    ------------------------ tokens --------------------------
242     var tSet : Set{Token} .
        var agents : [Configuration] .

        --- NOTE: it assumes that agents' sigmas have no
        ---       minted tokens in their domains (as it
247     ---       should be for)
        op (_).freeTokens : closedConfiguration ->
                                             Set{Free-Token} .
        eq (cGamma).freeTokens =
                     $freeTokens((cGamma).sigmas, empty) .
252
        op $freeTokens : [Configuration] Set{Token} ->
                                         Set{Token} .
        eq $freeTokens(none, tSet) = tSet .
        eq $freeTokens((agents < A0 : agState | * w >), tSet) =
257                       $freeTokens(agents,
                                  ( filterFree(dom(w) ), tSet)) .


    ------------------------ agents --------------------------
262     var A0 : Agent-Id .
        var agState : Agent-State .
        var attr : AttributeSet .
        var msg : Msg .
        var cGamma : closedConfiguration .
267     var aSet : Set{Agent-Id} .

        op _.agent_ : [Configuration] Agent-Id -> Object
```

```
                                        [prec 50] .
    var w : Map{Token, Float0+} .
272
    eq (none).agent(A0) = objectNotFound .
    ceq ((gamma obj)).agent(A0) = obj
                if < A0 : agState | * w > := obj .
    eq ((gamma obj)).agent(A0) =
277                             (gamma).agent(A0) [owise] .

    var port : Portal .
    op (_).agents : Configuration -> Set{Agent-Id} .
    eq (gamma).agents = $agents(gamma, empty) .
282
    op $agents : [Configuration] Set{Agent-Id} -> Set{Agent-Id} .
    eq $agents(none, aSet) = aSet .
    eq $agents(gamma < A0 : agState | attr >, aSet) =
            $agents(gamma, (aSet, A0) ) .
287 eq $agents(gamma obj, aSet) = $agents(gamma, aSet) .
    eq $agents(gamma msg, aSet) = $agents(gamma, aSet) .
    eq $agents(gamma port, aSet) = $agents(gamma, aSet) .

---------------------- wallets -------------------------
292 op _.sigma : Object -> Map{Token, Float0+}
                        [prec 20]  . --- 21 is []
    eq (< A0 : agState | * w >).sigma =  w .
    op _.sigma : Object -> Map{Token, Float0+}
                        [prec 20]  . --- 21 is []
297 --- NOTE: not used
    op isSigma : Object -> Bool .
    ceq isSigma(obj) = true
                if < A0 : agState | * w > := obj .
    eq isSigma(obj) = false [owise] .
302

    var sigmas : [Configuration] .

    op (_).sigmas : closedConfiguration -> [Configuration] .
307 eq (cGamma).sigmas = $sigmas((cGamma).config, none) .

    op $sigmas : [Configuration] [Configuration] ->
                                            [Configuration] .
    eq $sigmas(none, sigmas) = sigmas .
312 eq $sigmas(gamma < A0 : agState | attr >, sigmas) =
            $sigmas(gamma, sigmas < A0 : agState | attr >) .
    eq $sigmas(gamma obj, sigmas) = $sigmas(gamma, sigmas) .
    eq $sigmas(gamma msg, sigmas) = $sigmas(gamma, sigmas) .

317 ----------------- Params ---------------------------
    --- assumes there is only one LiqParams object
    op (_).params : Configuration -> [Configuration] .
    eq (gamma).params = $params(gamma) .

322 var paramObj : Params .
```

```
    op $params : [Configuration] -> [Configuration] .
    eq $params(none) = none .
    eq $params(gamma paramObj) = paramObj .
    eq $params(gamma obj) = $params(gamma) .
    eq $params(gamma msg) = $params(gamma) .
    eq $params(gamma port) = $params(gamma) .

    op (_).CMin : Configuration -> Float0+ .
    eq (gamma).CMin = ((gamma).params).CMin .

    op (_).Rliq : Configuration -> Float0+ .
    eq (gamma).Rliq = ((gamma).params).Rliq .
endm
    ⋮            ⋮            ⋮

mod BC-LP-MODEL is
    including CONFIGURATION-OBSERVERS .
    including MESSAGE-ID .
    ⋮            ⋮            ⋮

------------------------ Rules ----------------------------

-------------------------- DEPOSIT ------------------------
    var agState agState' : Agent-State .

    var pi' : Pi .
    vars c0 c1 : Float0+ .

    var r : Int .
    var tau : Free-Token .
    var v v' : Float0+ .
    var pi_f : Map{Token, Float0+} .
    var pi_l : Map{Agent-Id, Map{Token, Float0+}} .
    var pi_m : Map{Free-Token, Pair{Token, Float0+}} .
    vars p p' : Map{Token, Float0+} .

    var attrS : AttributeSet .
    var oid : Oid .
    var cid : Cid .
    var A0 : Agent-Id .
    var pi : Pi .

    op _.attributes : Object -> AttributeSet [prec 50] .
    eq (< oid : cid | attrS  >).attributes  = attrS .
    eq (obj).attributes = none .


    op deposit : Agent-Id Pair{Float0+, Token} -> Msg [ctor] .
    crl [deposit] :
       [ gamma
       < R(r) : Round | none >
       < C(r) : Coll | attr >
```

```
        < A0 : agState | * sigma >
938     (pi | p)
        deposit(A0, (v, tau)) ]
        => [ gamma
           < R(r + 1) : Round | none >
           < C(r + 1) : Coll | computeC(gamma', empty) >
943        < A0 : agState | * (sigma - v : tau)
                                + v' : (tau)' >
           (pi' | p) ]
     if sigma[tau] >= v /\
        {fund: pi_f, loan: pi_l, mint: pi_m} := pi /\
948     v' := v / ((pi).ER(tau)) /\
        pi' := {fund: (pi_f + v  : tau), loan: pi_l,
                mint: (pi_m + v' : tau) }  /\
        gamma' :=  gamma
           < A0 : agState | * (sigma - v : tau)
953                             + v' : (tau)' >
                 (pi' | p) .
     ⋮             ⋮             ⋮

  ----------------------- LIQUIDATE -------------------------
     op liquidate : Agent-Id Agent-Id
                    Pair{Float0+, Token}
1182                Minted-Token -> Msg [ctor] .

     var B0 : Agent-Id .
     vars sigmaA sigmaB
          sigmaA' sigmaB' : Map{Token, Float0+} .
1187
     crl [liquidate] :
        [ gamma                  --- i. represents the i-th
        < R(r) : Round | none > --- constraint in the liqui-
        < C(r) : Coll | attr >  --- date action def
1192    < A0 : agState | * sigmaA > --- (Figure 2.1)
        < B0 : agState' | * sigmaB >
        (pi | p)
        liquidate(A0, B0, (v, tau), tau') ]
        => [ gamma
1197       < R(r + 1) : Round | none >
           < C((r + 1)) : Coll | computeC(gamma', empty) >
           < A0 : agState | * sigmaA' >
           < B0 : agState' | * sigmaB' >
           (pi' | p) ]
1202
    if sigmaA[tau] >= v /\                          --- 2.
       (pi).loan[B0][tau] >= v /\                   --- 3.
       v' := v * (p[tau] / (p[(pi).u(tau')])) *     --- 4.
             (gamma).Rliq /\
1207   sigmaB[tau'] >= v' /\                        --- 5.

       pi' := {
           fund: (pi).fund + v : tau,               --- 6.
```

```
            loan: insert(B0,                                  --- 7.
                         ((pi).loan[B0] - v : tau), (pi).loan),
            mint: (pi).mint } /\
          sigmaA' := (sigmaA - v : tau) + v' : tau' /\        --- 8.
          sigmaB' := sigmaB - v' : tau' /\                    --- 9.
          gamma' := gamma
                    < A0 : agState | * sigmaA' >
                    < B0 : agState' | * sigmaB' >
                    (pi' | p) /\
          (gamma  < A0 : agState | * sigmaA >
                  < B0 : agState' | * sigmaB >
                  (pi | p) ).C(B0) < (gamma).CMin /\           --- 10.
          (gamma').C(B0) <= (gamma).CMin .                    --- 11.


  ----------------------- TRANSFER -------------------------
      op transfer : Agent-Id Agent-Id
                    Pair{Float0+, Token} -> Msg [ctor] .

      crl [transfer] :
         [ gamma
         < R(r) : Round | none >
         < C(r) : Coll | attr >
         < A0 : agState | * sigmaA >
         < B0 : agState' | * sigmaB >
         transfer(A0, B0, (v, tau)) ]
         => [ gamma
              < R(r + 1) : Round | none >
              < C(r + 1) : Coll | attr >
            < A0 : agState | * sigmaA - v : tau >
            < B0 : agState' | * sigmaB + v : tau > ]
      if sigmaA[tau] >= v .
      ⋮              ⋮              ⋮

  mod BC-LP-MODEL-MODIFIERS is
      including BC-LP-MODEL .
      var cGamma : closedConfiguration .
      var msg : Msg .
      var attrSet : AttributeSet .
      var oid : Oid .

      op _>_ : closedConfiguration Msg ->
                           closedConfiguration [prec 20] .
      eq cGamma > msg = [(cGamma).config msg] .

      var ill-gamma : [Configuration] .
      op _>_ : closedConfiguration [Configuration] ->
                           closedConfiguration [prec 20] .
      eq cGamma > ill-gamma = [ (cGamma).config ill-gamma ] .


      var gamma : [Configuration] .
```

```
    var newP oldP : Map{Token, Float0+} .
    var pi : Pi .
    --- MultiVeStA - new price injection
    op updatePrice(_,_) : Map{Token, Float0+}
                          closedConfiguration
                           -> closedConfiguration .
    eq updatePrice(newP, [(pi | oldP) gamma]) =
                              [(pi | newP) gamma] .

    op genLiqParams(_,_) : Float0+ Float0+ ->
                                Configuration .
    eq genLiqParams(cminV, rliqV) =
       < P(0) : LiqParams | CMin(cminV), Rliq(rliqV) > .

    op genCollObj(_) : closedConfiguration -> Object .
    eq genCollObj(cGamma) =
     < C(0) : Coll | computeC((cGamma).config, empty) > .

    vars cminV rliqV : Float0+ .
    op addLiqParams(_,_,_) : Configuration Float0+
                             Float0+ -> Configuration .
    eq addLiqParams(gamma, cminV, rliqV) =
          gamma < P(0) : LiqParams | CMin(cminV),
                                     Rliq(rliqV) > .

    op addLiqParams(_,_,_) : closedConfiguration Float0+
                             Float0+ ->
                                   closedConfiguration .
    eq addLiqParams(cGamma, cminV, rliqV) =
       [ addLiqParams((cGamma).config, cminV, rliqV) ] .


    op (_).replaceLiqParams(_,_) : closedConfiguration
                                   Float0+ Float0+
                                -> closedConfiguration .
    eq (cGamma).replaceLiqParams(cminV, rliqV) =
  [ ((cGamma).config).$replaceLiqParams(none,
                                        cminV, rliqV) ] .

    var gamma' : Configuration .
    var attrS : AttributeSet .
    op (_).$replaceLiqParams(_,_,_) : Configuration
                                      Configuration
                                      Float0+ Float0+
                                         -> Configuration .
    eq (none).$replaceLiqParams(gamma', cminV, rliqV) =
            addLiqParams(gamma', cminV, rliqV) .
    eq (gamma < P(0) : LiqParams | attrS >)
             .$replaceLiqParams(gamma', cminV, rliqV) =
       gamma < P(0) : LiqParams | CMin(cminV),
                                  Rliq(rliqV) >
       gamma' .
    eq (gamma ill-gamma)
```

```
                .$replaceLiqParams(gamma', cminV, rliqV) =
      (gamma)
      .$replaceLiqParams(ill-gamma gamma', cminV, rliqV) .
endm
```

Listing A.7: `configurations.maude` - Maude specifications of an LP configuration

A.6 Agents

This listing specifies the sorts and operators implementing the heuristic for a liquidator.

```
     ⋮              ⋮              ⋮

mod LOR-STRATEGY is
    including BC-LP-MODEL .
    including LIQUIDATOR-UTILS .
    including LOR-DATA-STRUCTURES .

--------------- param selection top checks -----------------

--- checks if loan given by seized2Repaid(value-to-liquidate)
--- can be repaid by LOR
    including HO-FILTER{Pair{Pair{Agent-Id, Float0+},
                             ColPair{Free-Token,
                                     Minted-Token}}} .

    var gamma : Configuration .
    var v : Float0+ .
    vars LED LOR LED' : Agent-Id .
    op (_).isLoanRepayable(_,_) : Configuration Float0+
                                  Agent-Id ->
                      Func{ Pair{Pair{Agent-Id, Float0+},
                                 ColPair{Free-Token,
                                         Minted-Token}},
                            Bool } .
    eq (gamma).isLoanRepayable(maxRep%, LOR)
          [ ((LED, v'), (tau : tau')) ] =
            ((gamma).agent(LOR)).sigma[tau] >=
              ( (seized2Repaid((gamma).lp, v', tau, tau',
                               (gamma).Rliq )) *
                    maxRep% ) .

    op (_).filterRepayable(_,_,_) :
                     Configuration Float0+ Agent-Id
                     List{Pair{Pair{Agent-Id, Float0+},
                               ColPair{Free-Token,
                                       Minted-Token}}} ->
                     List{Pair{Pair{Agent-Id, Float0+},
```

```
                                    ColPair{Free-Token,
                                            Minted-Token}}} .
    eq (gamma).filterRepayable(maxRep%, LOR, aFFMList) =
       filter (aFFMList,
                  (gamma).isLoanRepayable(maxRep%, LOR)) .


--- final multiplication by Maxliq or liquidation speed
--- factor
     op (_).multiplyBy(_) : Pair{Pair{Agent-Id, Float0+},
                                  ColPair{Free-Token,
                                          Minted-Token}}
                            Float0+ ->
                            Pair{Pair{Agent-Id, Float0+},
                                  ColPair{Free-Token,
                                          Minted-Token}} .
    eq ( ((LED, v), (tau : tau')) ).multiplyBy(maxRep%) =
                   ((LED, (v * maxRep%)), (tau : tau')) .


--------------------------- scenario a ----------------------
--- gets the liquidatable amount of tokens, the liquidated
--- is given by min(liquidatable, all-minted), where
--- all-minted is the total amount of tokens of a
--- specific type minted by the LED.
--- This applies only to scenario (a): C(LED) > Rliq
    var maxRep% : Float0+ .
    var lp : BC{LP} .
    var pi : Pi .

    op (_).liquidatable(_,_) : Configuration Agent-Id
                               Minted-Token -> Float0+ .
    eq (gamma).liquidatable(LED, tau') =
          (gamma).collateralRise(LED) /
            ((gamma).lp).V^m(tau') .

    op (_).collateralRise(_) : Configuration Agent-Id
                                    -> Float0+ .
    eq (gamma).collateralRise(LED) =
          (gamma).V^m(LED) -
              (gamma).collateralThreshold(LED) .

    op (_).collateralThreshold(_) : Configuration Agent-Id
                                       -> Float0+ .
    eq (gamma).collateralThreshold(LED) =
          ( ( (gamma).V^m(LED)   -
              ((gamma).Rliq * ((gamma).lp).V^l(LED)) ) /
              ((gamma).CMin - (gamma).Rliq) ) *
              (gamma).CMin .

    including COND-TERNARY{Float0+} .
    op (_).updateVIfGTRliq :  Configuration ->
                      Func{Pair{Pair{Agent-Id, Float0+},
```

```
                                    ColPair{Free-Token,
                                            Minted-Token}},
                               Pair{Pair{Agent-Id, Float0+},
                                    ColPair{Free-Token,
                                            Minted-Token}}} .
    ceq (gamma).updateVIfGTRliq
                [((LED, v), (tau : tau'))] =
       ( (LED, ((v > v') ? v' : v)), (tau : tau') )
       if (gamma).C(LED) >= (gamma).Rliq /\
          v' := (gamma).liquidatable(LED, tau') .
    eq (gamma).updateVIfGTRliq
                [((LED, v), (tau : tau'))] =
       ( (LED, v), (tau : tau') ) [owise] .

    var aFFMList : List{Pair{Pair{Agent-Id, Float0+},
                             ColPair{Free-Token,
                                     Minted-Token}}} .

    op (_).updateRepaidUpToCMin(_) : Configuration
                       List{Pair{Pair{Agent-Id, Float0+},
                                 ColPair{Free-Token,
                                         Minted-Token}}} ->
                       List{Pair{Pair{Agent-Id, Float0+},
                                 ColPair{Free-Token,
                                         Minted-Token}}} .
    eq (gamma).updateRepaidUpToCMin(aFFMList) =
       map (aFFMList, (gamma).updateVIfGTRliq) .


---------------------- wrap in 4-tuples -------------------
--- ( (Agent-Id, value-to-liquidate),
---   (loanTokenType : collateralTokenType) )
    including HO-MAP{Pair{Agent-Id, Pair{Token, Float0+}},
                     Pair{Pair{Agent-Id, Float0+},
                          ColPair{Free-Token,
                                  Minted-Token}}}.
    var tau : Free-Token .
    var tau' : Minted-Token .
    var v' : Float0+ .
    op (_).combine : Pair{Agent-Id, Token} ->
                 Func{Pair{Agent-Id, Pair{Token, Float0+}},
                      Pair{Pair{Agent-Id, Float0+},
                           ColPair{Free-Token,
                                   Minted-Token}}}.
    eq ((LED, tau)).combine[(LED, (tau', v'))] =
                    ((LED, v'), (tau : tau')) .
    eq ((LED, tau)).combine[(LED', (tau', v'))] = nil .

    var aTFPList : List{Pair{Agent-Id,
                         Pair{Token, Float0+}}} .

    including HO-MAP{Pair{Agent-Id, Token},
                     Pair{Pair{Agent-Id, Float0+},
```

```
                              ColPair{Free-Token,
                                      Minted-Token}}} .
300
    op (_).$combine : List{Pair{Agent-Id,
                                Pair{Token, Float0+}}} ->
                      Func{Pair{Agent-Id, Token},
                           Pair{Pair{Agent-Id, Float0+},
305                        ColPair{Free-Token,
                                   Minted-Token}}} .
    eq (aTFPList).$combine[(LED, tau)] =
        map (aTFPList, ((LED, tau)).combine) .

310 var aTPList : List{Pair{Agent-Id, Token}} .
    op combine(_,_) : List{Pair{Agent-Id, Token}}
                      List{Pair{Agent-Id,
                                Pair{Token, Float0+}}} ->
                      List{Pair{Pair{Agent-Id, Float0+},
315                        ColPair{Free-Token,
                                   Minted-Token}}} .
    eq combine(aTPList, aTFPList) =
        map (aTPList, (aTFPList).$combine) .

320
------------------- LIQ params selection -------------------
    var aSet : Set{Agent-Id} .
    op (_).selectLIQParams(_,_,_) : Configuration Agent-Id
                                    Set{Agent-Id} Float0+  ->
325                 Pair{Pair{Agent-Id, Float0+},
                         ColPair{Free-Token,
                                   Minted-Token}} .
    eq (gamma).selectLIQParams(LOR, aSet, maxRep%) =
      ((gamma).lp).seized2Repaid(
330
          findMax4TV^m(
            (gamma).filterRepayable(maxRep%, LOR,
              (gamma).updateRepaidUpToCMin(
                combine(
335               (gamma).getAFLTPairs( aSet, LOR ),
                  (gamma).getAMWTPairs( aSet )
                ) ) )

                    ), (gamma).Rliq ) .
340 endm
    ⋮            ⋮            ⋮

mod LIQUIDATOR is
    including LOR-STRATEGY .
    including COND-TERNARY{Float0+} .
    including PAIR{
377                 ColPair{Free-Token, Minted-Token},
                    ColPair{Agent-Id, Float0+}
                  } .
```

```
    vars LOR LED : Agent-Id .
382
    var gamma : Configuration .
    var sigma : Map{Token, Float0+} .

    var tau : Free-Token .
387 var tau' : Minted-Token .

    var b : Bool .
    vars v v' : Float0+ .

392 var LED-candidates : Set{Agent-Id} .
    vars vTmp s : Float0+ .

    crl [issue-liquidate] :
       [ gamma
397    < LOR : LIQ-Speed(s) | * sigma > ]
    --- DEL all previous liquidate actions issued by LOR
    --- over LED
    => [ filterMsgOut( gamma, (LIQ, (LOR, LED)) )
       < LOR : LIQ-Speed(s) | * sigma >
402    liquidate(LOR, LED, (v, tau), tau')
       ]
    if ([gamma]).isRewritten /\ --- this ensures that 2
                                --- liquidates can't be
                                --- issued
407    LED-candidates :=
        ( getUndercoll(gamma) \ getCollNegligible(gamma)) /\
       ((LED, vTmp), (tau : tau')) :=
           ( gamma
           < LOR : LIQ-Speed(s) | * sigma >
412        ).selectLIQParams(LOR, LED-candidates, s) /\
       ((LED, vTmp), (tau : tau')) =/= dummyMax4Tuple /\
       LOR =/= LED /\ --- agent can't liquidate itself
       --- WA : allows to liquidate everything in case
       ---      vTmp * its price is below a threshold, owise
417    ---      computation would not converge
       v := (((vTmp * ((gamma).lp).price[tau]) < s) ?
                 vTmp : (vTmp * s) ) .
endm
  ⋮             ⋮             ⋮
```

Listing A.8: `agents.maude` - Maude specifications of the heuristic modelling a rational liquidator

A.7 Integration with Maude LTL model checker

This listing specifies the sorts and operators implementing the integration with the Maude native LTL model checker, shipped in the library `model-checker.maude`.

```
                ⋮
mod BC-LP-PREDS is
     including BC-LP-MODEL-TEST .
     including MODEL-CHECKER .

     var illConfig illConfig' : [Configuration] .
     var gamma gamma0 gamma* : closedConfiguration .
     var config config* : Configuration .
     var tau : Token .
     var tSet : Set{Token} .
     vars curS v : Float0+ .
     var obj : Object .
     var msg : Msg .
     var bal : Map{Token, Float0+} .
     var agState : Agent-State .

     op testLemma1 : closedConfiguration -> Bool .
     eq testLemma1(gamma) =
              $testLemma1((gamma).config,
                          ((gamma).config).mintedTokens) .

     var mTokenSet : Set{Minted-Token} .
     var tau' : Minted-Token .

     --- pre: 1. Set{Minted-Token} is (config).mintedTokens
     ---      2. minted token tau is tau'
     op $testLemma1 : Configuration Set{Token} -> Bool .
     eq $testLemma1(config, empty) = true .
     eq $testLemma1(config, tau') =
            (config).balSum(tau') ==
                     snd((config).pi.m[(config).u(tau')]) .
     eq $testLemma1(config, (mTokenSet, tau')) =
         if ((config).balSum(tau') ==
              snd((config).pi.m[(config).u(tau')])) then
            $testLemma1(config, (mTokenSet))
         else
            false
         fi .

     op (_).balSum(_) : Configuration Token -> Float0+ .
     eq (config).balSum(tau) = $balSum(config, tau, 0.0) .

     op $balSum : [Configuration] Token Float0+ ->
                                                  Float0+ .
     eq $balSum(none, tau, curS) = curS .
     eq $balSum((illConfig < A0:Agent-Id : agState | * bal >),
                          tau, curS) =
                $balSum(illConfig, tau, bal[tau] + curS) .
     eq $balSum((illConfig obj), tau, curS) =
                    $balSum(illConfig, tau, curS) .
     eq $balSum((illConfig msg), tau, curS) =
                    $balSum(illConfig, tau, curS) .
```

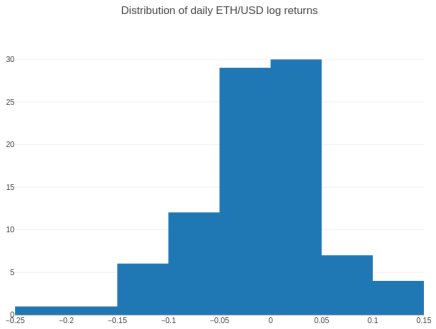
```
vars gamma0S curSpl g0LentERs g0NotLentERs curERs :
                                Map{Token, Float0+} .

--- NOTE: this is not the ER of all free-tokens, but
---       only of those free-tokens which are being
---       lent.
--- FIXME: note that over here Minted-Token should
---        be considered NOT Free-Token, as it currently
---        is.
op lentERs : closedConfiguration ->
                          Map{Token, Float0+} .
eq lentERs(gamma) = $ERs( (gamma).config,
                          getLentTokens(gamma),
                          emptyM ) .

op notLentERs : closedConfiguration ->
                          Map{Token, Float0+} .
eq notLentERs(gamma) = $ERs( (gamma).config,
             (gamma).freeTokens \ getLentTokens(gamma),
             emptyM ) .

op $ERs : Configuration Set{Token}
          Map{Token, Float0+} ->
               Map{Token, Float0+} .
eq $ERs(config, empty, curERs) = curERs .
eq $ERs(config, tau, curERs) =
     insert(tau, (((config).lp).pi).ER(tau), curERs) .
eq $ERs(config, (tSet, tau), curERs) =
   $ERs(config, tSet,
     insert(tau, (((config).lp).pi).ER(tau), curERs)) .


op supply : closedConfiguration ->
                   Map{Token, Float0+} .
eq supply(gamma) = $supply((gamma).config,
                           (gamma).freeTokens,
                           emptyM ) .

op $supply : Configuration Set{Token}
             Map{Token, Float0+} ->
                  Map{Token, Float0+} .
eq $supply(config, empty, curSpl) = curSpl .
eq $supply(config, tau, curSpl) =
   insert(tau, (config).balSum(tau) +
               (config).pi.f[tau], curSpl) .
eq $supply(config, (tSet, tau), curSpl) =
   $supply(config, tSet, insert(tau,
               (config).balSum(tau) +
               (config).pi.f[tau], curSpl)) .


op filterGZ : Map{Token, Float0+} -> Set{Token} .
```

```
      eq filterGZ(bal) = $filterGZ(bal, empty) .

      op $filterGZ : Map{Token, Float0+} Set{Token} ->
                                                Set{Token} .
      eq $filterGZ(emptyM, tSet) = tSet .
      eq $filterGZ((bal ; tau |-> v), tSet) =
                            if v > 0.0 then
                               $filterGZ(bal, (tau, tSet))
                            else
                               $filterGZ(bal, tSet)
                            fi .

      var aSet : Set{Agent-Id} .
      var A0 : Agent-Id .
      var pi_l : Map{Agent-Id, Map{Token, Float0+}} .
      op getLentTokens : closedConfiguration
                                      -> Set{Token} .

      ceq getLentTokens(gamma) = $getLentTokens(
                                    pi_l, dom(pi_l), empty)
      if pi_l := ((gamma).config).pi.l .
      --- NOTE: it could be speeded up by passing here the
      ---       set of all free-tokens and adding a check
      --- pre: agents in aSet must be in domain of pi_l
      op $getLentTokens : Map{Agent-Id, Map{Token, Float0+}}
                          Set{Agent-Id} Set{Token}
                                              -> Set{Token} .
      eq $getLentTokens(pi_l, empty, tSet) = tSet .
      eq $getLentTokens(pi_l, (aSet, A0), tSet) =
             $getLentTokens(pi_l, (aSet),
                            (filterGZ(pi_l[A0]), tSet)) .

      var eps : Float0+ .

      subsort closedConfiguration < State .
      op lemma1 : -> Prop .
      eq gamma |= lemma1 = testLemma1(gamma) .


      --- lemma3
      op sameSupply : Map{Free-Token, Float0+} Float0+ ->
                                                Prop .
      eq gamma |= sameSupply(gamma0S, eps) =
                          supply(gamma) ~= gamma0S : eps .
endm
```

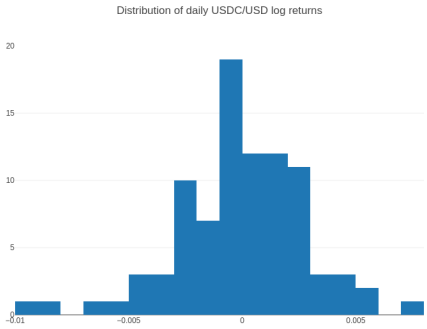
Listing A.9: `ltl.maude` - Integration of the Maude specification with the `MODEL-CHECKER` module

B Additional figures

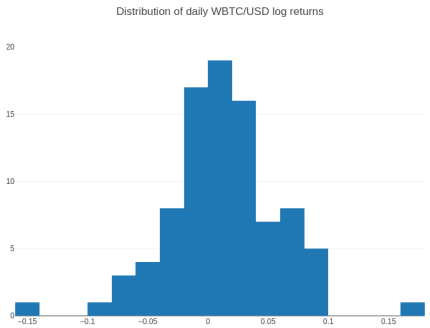
B.1 Modelling stock prices



(a) 13/01/2018-14/04/2018

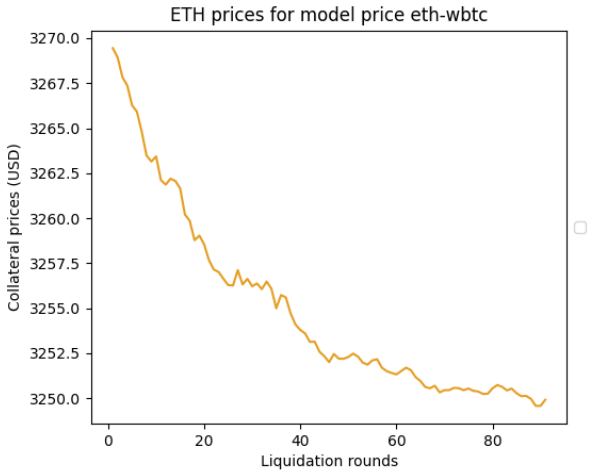


(b) 01/01/2020-01/04/2020

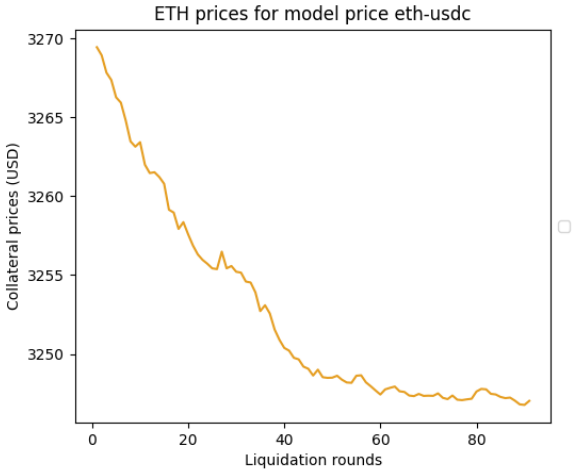


(c) 24/11/2020-23/02/2021

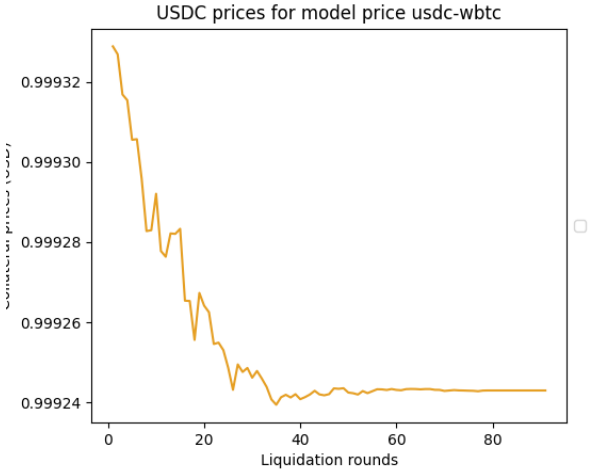
Figure B1: Distributions of daily log returns computed from trimester closing prices



(a)

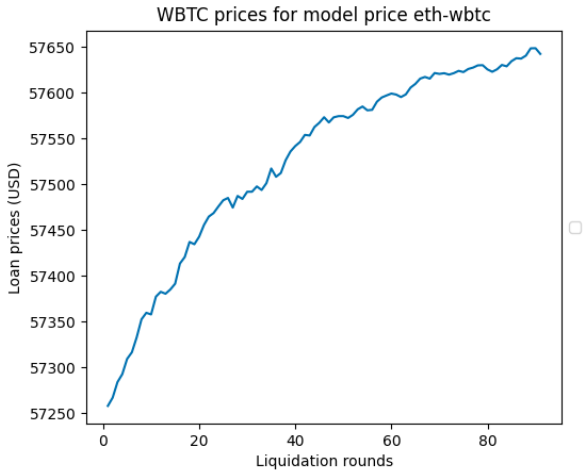


(b)

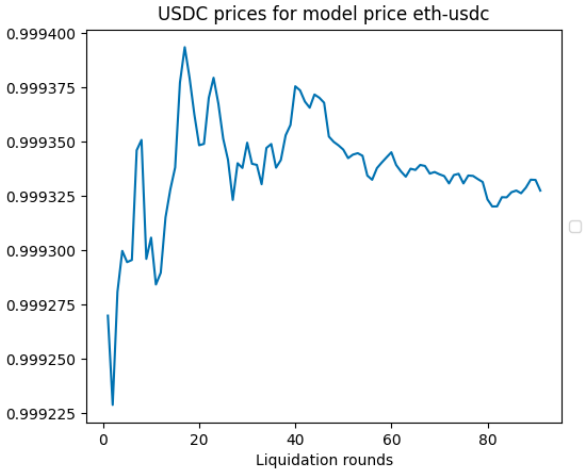


(c)

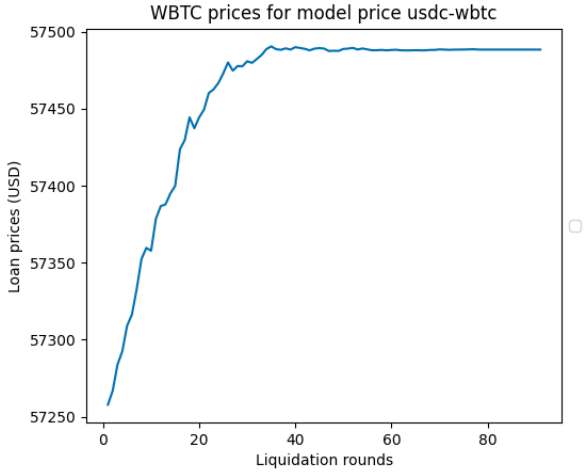
Figure B2: Prices predictions for the collateral assets produced, for each scenario in Table 4.1, by GBMs instantiated with the parameters in Table 4.2.



(a)



(b)



(c)

Figure B3: Prices predictions for the loan assets produced, for each scenario in Table 4.1, by GBMs instantiated with the parameters in Table 4.2.

B.2 Results of the statistical analysis

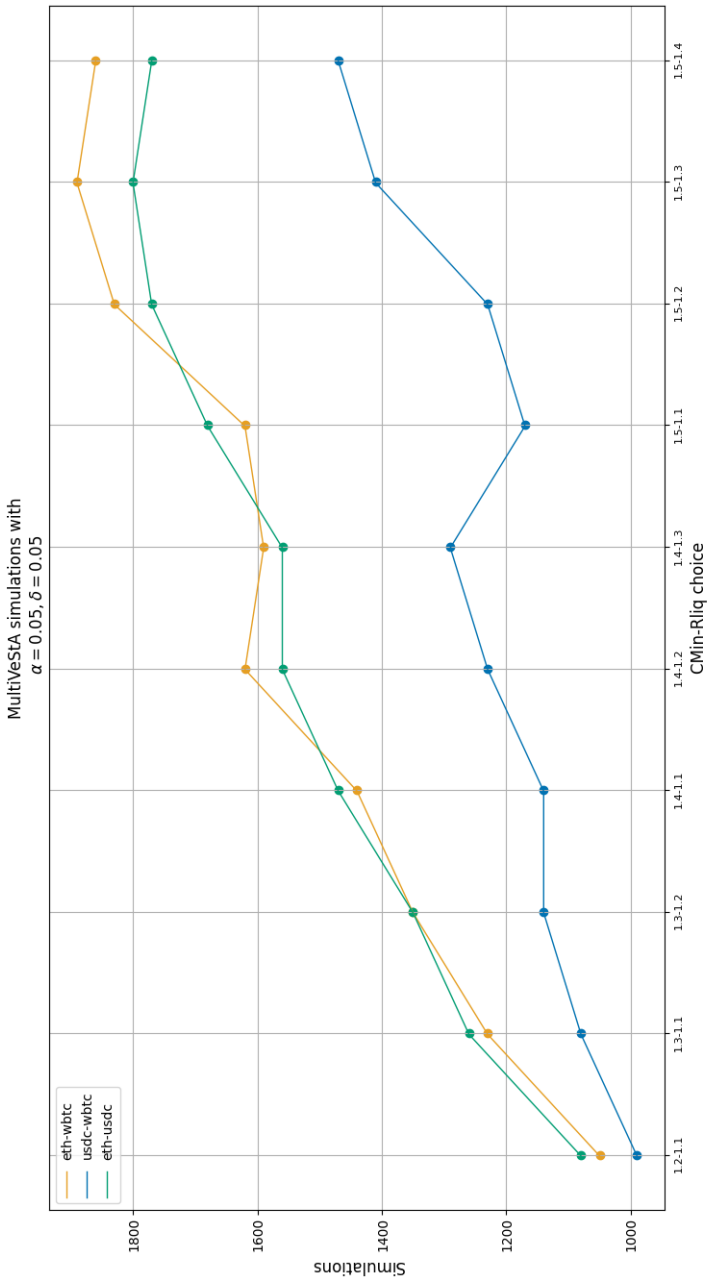
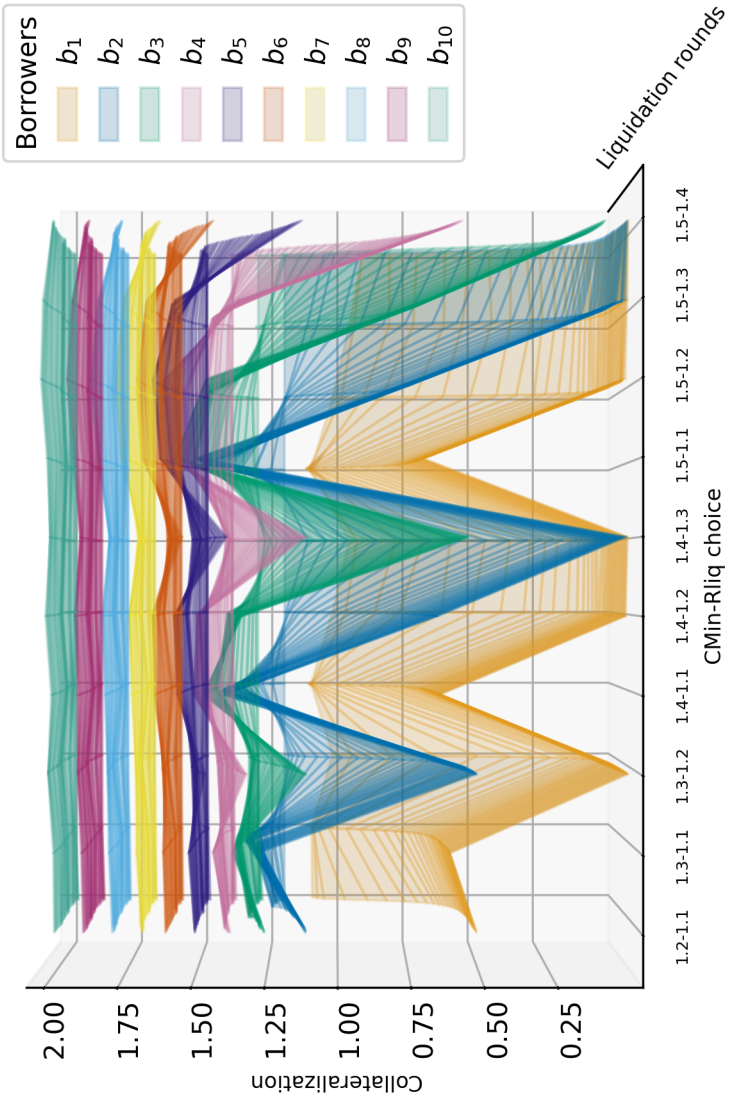
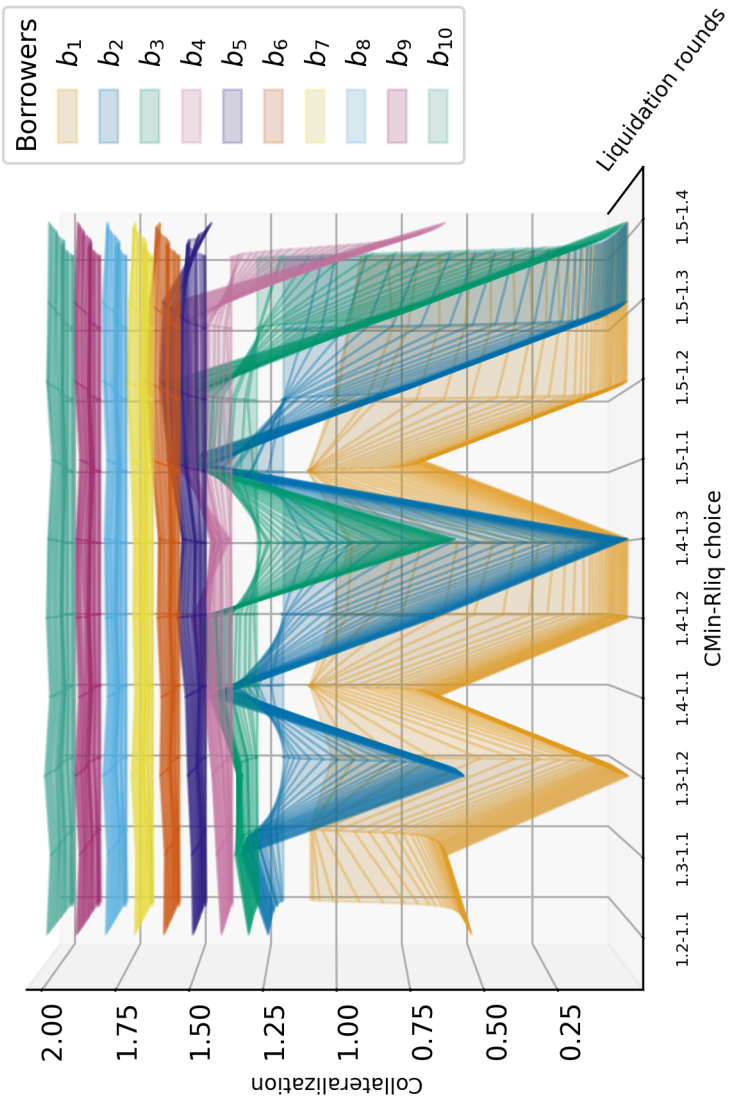


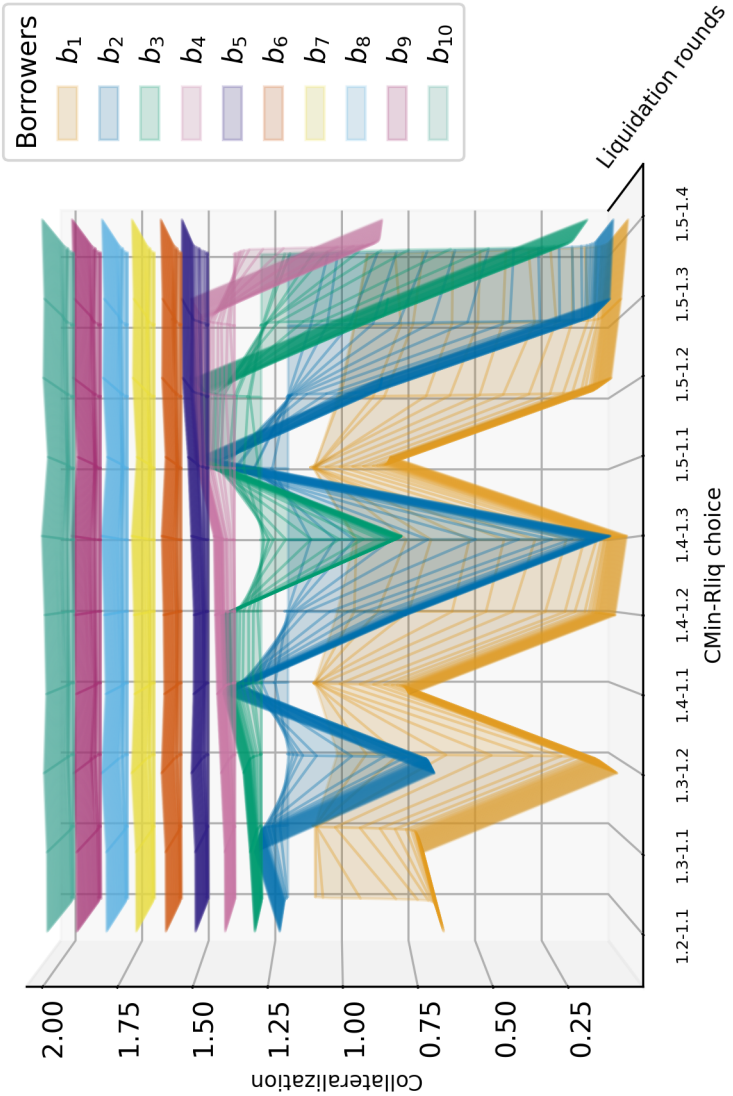
Figure B4: Number of LP simulations, per `CMin-rliq` choices, performed to obtain an acceptable approximation of the per-agent collateralization (Figures 5.4, 5.5 and B5).



(a) Scenario: eth-wbtc.



(b) Scenario: eth-usdc.



(c) Scenario: usdc-wbtc.

Figure B5: Per-borrower collateralization in the three prices scenarios, with varying liquidation rounds and `CMin-rliq` choices.

C Lending pools simulation

C.1 Full running example

```
search in SEARCH-EXAMPLE :
       Gamma^i0 =>* X:closedConfiguration .

Solution 1 (state 0) -- Γ^i_0, in Table 2.3
states: 1  rewrites: 147 in 0ms cpu (0ms real)
(~ rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ; tau(1) |-> 1.95e+2,
          loan: A |-> tau(1) |-> 8.0e+1 ;
                B |-> tau(1) |-> 1.0e+2 ;
                C |-> tau(1) |-> 1.25e+2,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 1.0e+2) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 1.0e+2) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
                  tau(0)' |-> 1.0e+2) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                   tau(1) |-> 5.0e+2 ;
                  tau(1)' |-> 5.0e+2) >
< C(0) : Coll | * (A,1.25),
                * (B,1.0),
                * (C,0.8),
                * (D,-) >
< R(0) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, A, (5.0e+1,tau(1)), tau(0)')
liquidate(D, B, (9.0909090909090907e+1,tau(1)), tau(0)')
liquidate(D, C, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 2 (state 1) -- Γ_{1,1}, in Table 2.3
states: 2  rewrites: 1247 in 0ms cpu (1ms real)
(~ rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ; tau(1) |-> 2.45e+2,
          loan: A |-> tau(1) |-> 3.0e+1 ;
                B |-> tau(1) |-> 1.0e+2 ;
```

```
                  C |-> tau(1) |-> 1.25e+2,
            mint: tau(0) |-> (tau(0)',3.0e+2) ;
                  tau(1) |-> (tau(1)',5.0e+2)
          }
 |
          tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 4.4999999999999993e+1) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 1.0e+2) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
                  tau(0)' |-> 1.0e+2) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                   tau(1) |-> 4.5e+2 ;
                  tau(0)' |-> 5.5000000000000007e+1 ;
                  tau(1)' |-> 5.0e+2) >
< C(1) : Coll | * (A,1.5),
                * (B,1.0),
                * (C,0.8),
                * (D,-) >
< R(1) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, B, (9.0909090909090907e+1,tau(1)), tau(0)')
liquidate(D, C, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 3 (state 2) -- Γ1,2, in Table 2.3
states: 3  rewrites: 2335 in 0ms cpu (1ms real)
(~ rewrites/second)
X:closedConfiguration --> [(
          {
            fund: tau(0) |-> 3.0e+2 ;
                  tau(1) |-> 2.8590909090909088e+2,
            loan: A |-> tau(1) |-> 8.0e+1 ;
                  B |-> tau(1) |-> 9.0909090909090935 ;
                  C |-> tau(1) |-> 1.25e+2,
            mint: tau(0) |-> (tau(0)',3.0e+2) ;
                  tau(1) |-> (tau(1)',5.0e+2)
          }
 |
          tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 1.0e+2) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 0.0) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
```

```
                    tau(0)' |-> 1.0e+2) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 4.0909090909090912e+2 ;
                  tau(0)' |-> 1.0e+2 ;
                  tau(1)' |-> 5.0e+2) >
< C(1) : Coll | * (A,1.25),
                * (B,0.0),
                * (C,0.8),
                * (D,-) >
< R(1) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, A, (5.0e+1,tau(1)), tau(0)')
liquidate(D, C, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 4 (state 3) -- Γ1,3, in Table 2.3
states: 4  rewrites: 3423 in 0ms cpu (2ms real)
(~ rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 2.8590909090909088e+2,
          loan: A |-> tau(1) |-> 8.0e+1 ;
                B |-> tau(1) |-> 1.0e+2 ;
                C |-> tau(1) |-> 3.4090909090909093e+1,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 1.0e+2) >
< B : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 1.0e+2) >
< C : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.25e+2 ;
                  tau(0)' |-> 0.0) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 4.0909090909090912e+2 ;
                  tau(0)' |-> 1.0e+2 ;
                  tau(1)' |-> 5.0e+2) >
< C(1) : Coll | * (A,1.25),
                * (B,1.0),
                * (C,0.0),
                * (D,-) >
< R(1) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, A, (5.0e+1,tau(1)), tau(0)')
liquidate(D, B, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 5 (state 4) -- Γ2,1, in Table 2.3
```

```
states: 5  rewrites: 4575 in 3ms cpu (3ms real)
(1374699 rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 3.3590909090909088e+2,
          loan: A |-> tau(1) |-> 3.0e+1 ;
                B |-> tau(1) |-> 9.0909090909090935 ;
                C |-> tau(1) |-> 1.25e+2,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                   tau(0)' |-> 4.4999999999999993e+1) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                   tau(0)' |-> 0.0) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
                   tau(0)' |-> 1.0e+2) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                   tau(1) |-> 3.5909090909090912e+2 ;
                   tau(0)' |-> 1.55e+2 ;
                   tau(1)' |-> 5.0e+2) >
< C(2) : Coll | * (A,1.4999999999999998),
                * (B,0.0),
                * (C,8.0000000000000004e-1),
                * (D,-) >
< R(2) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, C, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 6 (state 5) -- Γ2,2, in Table 2.3
states: 6  rewrites: 5727 in 3ms cpu (4ms real)
(1720853 rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 3.3590909090909088e+2,
          loan: A |-> tau(1) |-> 3.0e+1 ;
                B |-> tau(1) |-> 1.0e+2 ;
                C |-> tau(1) |-> 3.4090909090909093e+1,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
```

```
                    tau(1) |-> 8.0e+1 ;
                    tau(0)' |-> 4.4999999999999993e+1) >
< B : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.0e+2 ;
                    tau(0)' |-> 1.0e+2) >
< C : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.25e+2 ;
                    tau(0)' |-> 0.0) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 3.5909090909090912e+2 ;
                    tau(0)' |-> 1.55e+2 ;
                    tau(1)' |-> 5.0e+2) >
< C(2) : Coll | * (A,1.4999999999999998),
                * (B,1.0),
                * (C,0.0),
                * (D,-) >
< R(2) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, B, (9.0909090909090907e+1,tau(1)), tau(0)')]

Solution 7 (state 6) -- Γ2,3, in Table 2.3
states: 7  rewrites: 7436 in 6ms cpu (7ms real)
(1120385 rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 3.7681818181818176e+2,
          loan: A |-> tau(1) |-> 8.0e+1 ;
                B |-> tau(1) |-> 9.0909090909090935 ;
                C |-> tau(1) |-> 3.4090909090909093e+1,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 8.0e+1 ;
                    tau(0)' |-> 1.0e+2) >
< B : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.0e+2 ;
                    tau(0)' |-> 0.0) >
< C : noState | * (tau(0) |-> 0.0 ;
                    tau(1) |-> 1.25e+2 ;
                    tau(0)' |-> 0.0) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                    tau(1) |-> 3.1818181818181824e+2 ;
                    tau(0)' |-> 2.0e+2 ;
                    tau(1)' |-> 5.0e+2) >
< C(2) : Coll | * (A,1.25),
                * (B,0.0),
                * (C,0.0),
                * (D,-) >
```

```
< R(2) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1000000000000001) >
liquidate(D, A, (5.0e+1,tau(1)), tau(0)')]

Solution 8 (state 7) -- Γ3,1, in Table 2.3
states: 8  rewrites: 9701 in 9ms cpu (10ms real)
(974093 rewrites/second)
X:closedConfiguration --> [(
        {
          fund: tau(0) |-> 3.0e+2 ;
                tau(1) |-> 4.2681818181818176e+2,
          loan: A |-> tau(1) |-> 3.0e+1 ;
                B |-> tau(1) |-> 9.0909090909090935 ;
                C |-> tau(1) |->
    3.4090909090909093e+1,
          mint: tau(0) |-> (tau(0)',3.0e+2) ;
                tau(1) |-> (tau(1)',5.0e+2)
        }
 |
        tau(0) |-> 1.0 ; tau(1) |-> 1.0
)
< A : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 8.0e+1 ;
                  tau(0)' |-> 4.4999999999999993e+1) >
< B : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.0e+2 ;
                  tau(0)' |-> 0.0) >
< C : noState | * (tau(0) |-> 0.0 ;
                   tau(1) |-> 1.25e+2 ;
                  tau(0)' |-> 0.0) >
< D : noState | * (tau(0) |-> 1.0e+2 ;
                   tau(1) |-> 2.6818181818181824e+2 ;
                  tau(0)' |-> 2.55e+2 ;
                  tau(1)' |-> 5.0e+2) >
< C(3) : Coll | * (A,1.5),
                * (B,0.0),
                * (C,0.0),
                * (D,-) >
< R(3) : Round | none >
< P(0) : LiqParams | CMin(1.5),Rliq(1.1) >]

No more solutions.
states: 8  rewrites: 10827 in 9ms cpu (11ms real)
(1087157 rewrites/second)
```

Listing C.1: `search.maude` - Full running example representing the configurations in Table 2.3